UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

UMA LINGUAGEM DE PROTOCOLOS PARA DESCREVER SISTEMAS DISTRIBUÍDOS TOLERANTES A FALHAS

Daniela Pedro Henriques

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA

Especialização em Arquitectura, Sistemas e Redes de Computadores

2014

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

UMA LINGUAGEM DE PROTOCOLOS PARA DESCREVER SISTEMAS DISTRIBUÍDOS TOLERANTES A FALHAS

Daniela Pedro Henriques

DISSERTAÇÃO

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

Dissertação orientada pelo Prof. Doutor Vasco Manuel Thudichum de Serpa Vasconcelos
e co-orientado pelo Prof. Doutor Alysson Neves Bessani

2014

Agradecimentos

Em primeiro lugar, quero agradecer à minha família. Aos meus pais, pelo apoio incondicional e por tornarem possíveis estes meus anos académicos. Eles sentiram um pouco as dificuldades por que passei e partilharam as alegrias. Sei que estão muito orgulhosos, por eu estar a terminar a minha vida académica e a começar uma nova etapa.

Quero agradecer ao meu namorado. Por todo o apoio que me deu, foi uma grande ajuda nestes anos académicos, em que partilhou comigo os seus conhecimentos e me ajudou a ultrapassar as dificuldades sempre com uma palavra positiva. Obrigado por tudo.

De seguida, quero agradecer aos meus amigos. À Daniela e à Mafalda por me aturarem e me animarem sempre que eu precisei. Passámos por muitos momentos juntas e sei que posso sempre contar com elas quando precisar. Obrigado por me estarem sempre a perguntar se já terminei a tese, para poder ter tempo para estar convosco. Agora é o momento!

Aos meus orientados, Professor Vasco Vasconcelos e Professor Alysson Bessani, um muito obrigado por toda a ajuda que me deram. Quero agradecer pelo tempo que dispensaram para me orientar, por estarem sempre prontos para trocar ideias comigo, sempre de forma aberta e comunicativa, sem qualquer crítica negativa.

Ao futuro.

Resumo

Nos sistemas distribuídos tolerantes a falhas existem vários processos que comunicam entre si que podem falhar a qualquer momento, interferindo com o curso normal da comunicação, o que torna estes sistemas mais complexos e difíceis de implementar. Quando se escreve este código complexo todas as operações de comunicação têm que ser definidas de modo a que os processos saibam que operações executar, os tipos de mensagens e os processos envolvidos nas operações. Atualmente não existe nenhuma técnica comum para escrever este tipo de programas, assim cada programador usa as suas próprias técnicas, tornando este processo mais lento e difícil.

Hoje em dia, são usadas diferentes abordagens para verificar que um programa que envolve trocas de mensagens está bem definido e funciona para cada situação possível. A abordagem mais comum consiste em passar o programa por uma série de testes criados pelo programador, o que pode demorado e pouco fiável pois é difícil testar todos os casos possíveis.

Esta tese sugere uma abordagem que se certifica que o programa troca mensagens por uma ordem pré-estabelecida, mesmo na presença de falhas que afetam estas trocas, e que poderá tornar a verificação destes programas mais rápida e mais precisa. A nossa abordagem consiste em especificar todas as interações de comunicação através de uma linguagem de protocolos que descreve programas distribuídos tolerantes a falhas.

Esta linguagem é usada para construir protocolos globais que especificam toda a comunicação do sistema. Através de regras de tradução, os protocolos globais são projetados para protocolos locais que especificam a comunicação do ponto de vista de cada processo. Por fim, é criado um programa que implementa as ações de cada um dos processos participantes no sistema. As operações de comunicação nestes programas seguem uma API da linguagem *Erlang* que definimos e que fornece uma forma comum de representar o envio/receção de mensagens.

Usando esta API poderá ser possível futuramente fazer uma correspondência entre os protocolos locais e os programas de cada processo, de modo a verificar em tempo de compilação que os programas trocam mensagem por uma ordem correta.

Palavras-chave: sistemas distribuídos, tolerância a falhas, trocas de mensagens, especificação de protocolos

Abstract

Fault-tolerant distributed systems handle a variety of communication interactions between multiple participants that can fail at any moment, interfering with the normal flow of communications and therefore making the systems more complex to implement. When writing this complex code all communication operations need to be defined, so that participants know which operations they have to execute, the message types involved in the operations and who participates in the operations. Until today there is no common strategy to write this type of programs, so each programmer uses his own techniques, making the process slower and more difficult.

Nowadays, to make sure that a message-passing program is well defined and works in every situation, it is possible to use different approaches. The most common consisting in passing a program through a series of tests, which can be time-consuming and lacks accuracy due to the difficulty of covering all possible cases.

This thesys suggests an approach that makes sure a program exchanges messages in a pre-established order even in the presence of failures that affect the message-passing interactions. Besides, it could make the process of verifying a program faster and more accurate. It consists in specifying all the communication interactions using a language of protocols to describe fault-tolerant distributed systems.

This language is used to create global protocols that specify all the communication of a system. Then translation rules are used to project global protocols into local protocols that specify the communication point of view of a process. Finally, for each process is created an individual program that implements its actions. The communication operations in these programs follow an API of the language Erlang that we created, which offers a common model to represent message-passing.

In the future, we could use this API to find connections between the local protocols and the programs of each process that would help verify at compile time that the communication of the program is well defined.

Keywords: fault tolerance, distributed systems, message passing, protocol specification

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Implementar sistemas distribuídos tolerantes a falhas é uma tarefa complexa. Os programas que envolvem comunicação por mensagens apresentam vários problemas, podem falhar facilmente quando recebem uma mensagem que não reconhecem ou que se encontra fora de ordem, podem não enviar uma mensagem na altura correta, os processos podem falhar o que pode levar a que processos dependentes do processo que falhou fiquem bloqueados. Dois processos podem ficar bloqueados porque ambos ficam à espera de receber uma mensagem, entre tantos outros problemas. Todos estes problemas têm que ser tratados pelos programadores durante e após a implementação. Deve haver um emparelhamento entre as operações de comunicação, por exemplo um envio tem que ter uma receção correspondente, há que ter em atenção a ordem das mensagens, tem que se lidar com a falha dos processos.

Atualmente para encontrar erros na comunicação são usadas maioritariamente técnicas que atuam em tempo de execução. A técnica mais comum consiste em executar uma série de testes criados pelo programador para verificar se o programa está construido corretamente. No entanto, criar todos esses testes é uma tarefa lenta e não completa, pois é impossível cobrir todas as interações de comunicação que ocorrem durante a execução do programa. Outra técnica para encontrar estes erros consiste na verificação de modelos [2], no qual o espaço de estados do sistema é explorado exaustivamente e de forma automatizada, mas esta técnica também apresenta alguns problemas.

Para contornar os problemas da verificação em tempo de execução e para conseguir encontrar este erros o mais cedo possível pode-se detetá-los em tempo de compilação. Para tal é necessário preparar uma especificação explícita da comunicação do sistema, que inclua todas as interações possíveis. Baseados nestas observações decidimos criar uma linguagem que permite descrever protocolos que especifiquem toda a comunicação do sistema, cobrindo todas as interações, incluindo os casos em que os processos falham. Os protocolos criados por esta linguagem podem ser usados para aumentar o poder do compilador, eliminando a maioria dos erros de comunicação. Combinado com alguma técnica de verificação em tempo de execução é possível verificar que os sistemas toleran-

tes a falhas estão bem implementados.

O nosso projeto foca-se na criação da linguagem para descrever os protocolos globais e na tradução destes protocolos para protocolos que especificam a comunicação de cada participante. Também apresentarmos um modelo para programar sistemas distribuídos tolerantes a falhas, mas não é nosso objetivo criar um verificador que permita validar programas de encontro aos protocolos.

1.1 Objetivos

Este projeto tem como objetivo especificar sistemas distribuídos tolerantes a falhas, facilitando a sua implementação. Para alcançar este objetivo criámos uma linguagem para especificar padrões na comunicação. Esta linguagem permite descrever protocolos globais que representam toda a comunicação do sistema de uma forma centralizada. Para além disso analisámos a semântica dos protocolos (como por exemplo o que cada operação de comunicação representa), de modo a assegurarmo-nos que os protocolos representam corretamente sistemas distribuídos tolerantes a falhas.

Definimos também regras de tradução que permitem fazer a projeção dos protocolos globais para protocolos locais que representam a comunicação do ponto de vista local a cada um dos participantes do protocolo global. Para além disso iremos definimos uma API (*Application Programming Interface*) de comunicação, fornecendo assim um modelo para implementar os sistemas distribuídos tolerantes a falhas, facilitando desta forma a sua implementação.

Para cada participante é criado um programa individual que utiliza a API de comunicação. Comparando as funções da API usadas na implementação com os protocolos locais do participante, pode-se detetar uma correspondência que é a chave para a verificação do programa em tempo de compilação.

1.2 Observações sobre o planeamento

Relativamente ao planeamento definido no relatório preliminar houve um atraso de cerca de dois meses devido a: (1.) refinar constantemente a linguagem, que sofreu diversas alterações até conseguirmos especificar devidamente os casos de estudo que apresentámos, (2.) investigar uma forma de implementar os sistemas que fosse geral e estivesse de acordo com os protocolos locais e (3.) a aprendizagem da linguagem *Erlang*. Foram os ligeiros atrasos em algumas etapas que levaram ao atraso geral da conclusão da tese, mas que não afetaram em todo a conclusão das tarefas definidas.

1.3 Estrutura do documento

Este documento está organizado nos seguintes capítulos:

2. Trabalho relacionado - Apresenta várias abordagem que oferecem soluções para os problemas na implementação de sistemas distribuídos tolerantes a falhas e linguagens de programação que estão relacionadas com o nosso trabalho.

3. Uma linguagem de protocolos para descrever sistemas distribuídos tolerantes a falhas - Apresenta linguagem de protocolos e explica através de um exemplo como se definem protocolos globais. Também apresenta a linguagem de protocolos locais e as regras de tradução global-local. Usando o mesmo exemplo, mostra a construção dos protocolos locais correspondentes.

4. Casos de estudo - Apresenta dois casos de estudo para os quais são construídos protocolos globais e locais.

5. Implementação - Apresenta a API de comunicação criada por nós e mostra excertos da implementação de protocolos usando a API.

6. Conclusão.

Capítulo 2

Trabalho relacionado

Este capítulo divide-se em duas partes, cada uma debruçando-se sobre um conceito diferente: as dificuldades na implementação de sistemas distribuídos tolerantes a falhas, e as linguagens de descrição de protocolos usadas para representar as interações entre processos em aplicações distribuídas.

2.1 Dificuldades na implementação de sistemas distribuídos tolerantes a falhas

Nesta secção iremos descrever várias abordagens que facilitam a implementação de sistemas distribuídos tolerantes a falhas. Iremos começar por introduzir um modelo baseado em regras [13], que ajuda os programadores na estruturação do seu código. De seguida iremos apresentar uma abordagem baseada em verificação de modelos [2], que permite definir todos os estados possíveis que uma comunicação pode originar, permitindo verificar durante a execução se o programa não se afasta desses estados. Outra abordagem consiste em testar o programa, de forma a verificar se o programa faz o que é esperado.

2.1.1 Um modelo para descrever sistemas distribuídos tolerantes a falhas

Geralmente cada programador usa as suas técnicas ao escrever código distribuído tolerante a falhas, mas o uso de um modelo para escrever este tipo de código complexo iria facilitar este processo, tornando-o mais rápido e criando um conhecimento partilhado entre os programadores. Na *RAMCloud* [13], um sistema de armazenamento de dados em centro de dados tolerante a falhas localizado em *clusters* com milhares de máquinas, também existiam estes problemas.

Para ultrapassá-los usa-se um modelo dividido em três camadas: regras, tarefas e *pools*. Este modelo consiste num conjunto de regras que podem ser desencadeadas em qualquer ordem e que são selecionadas para execução de acordo com o estado do sistema.

Cada regra descreve uma ação que será tomada quando a condição da regra for satisfeita, onde uma ação é um pedaço de código e a condição é um predicado sobre as variáveis de estado. Num código baseado em regras as ações são pequenas e não bloqueantes para que o controlo de fluxo dentro de uma regra seja previsível e que falhas não afetem a execução de uma ação. Para além disso, as mudanças de estado determinam a ordem de execução das regras, tornando a ordem imprevisível. Desta forma a execução adapta-se automaticamente na presença de falhas e concorrência.

Num sistema complexo o número de regras pode ser enorme, por isso estas são agrupadas em tarefas representadas como instâncias de classes, compostas por um registo do estado, um conjunto de regras e um objetivo. Numa tarefa as regras são especificadas como um conjunto de condições *if/else* contidas num único método, este método é invocado para aplicar as regras ao estado da tarefa. Quando a condição de uma regra é satisfeita a ação pode alterar o estado, levando a que as condições de outras regras sejam satisfeitas.

Por fim, as tarefas são agrupadas em *pools*, criando um subsistema. As *pools* dividem as tarefas em conjuntos independentes e que executam concorrentemente, isolando os subsistemas uns dos outros. Para reduzir a *overhead* da aplicação de regras as tarefas são divididas em ativas e inativas, sendo que só as regras de tarefas ativas são avaliadas. Uma tarefa mantém-se ativa até atingir o seu objetivo, nesse ponto torna-se inativa. As tarefas podem ser reativadas por uma mudança de estado que faz com que o seu objetivo deixe de ser satisfeito.

Na *RAMCloud* cada *pool* tem uma única *thread* que circula pelas tarefas ativas executando as suas regras, o que leva a uma serialização da execução. Desta forma remove-se a necessidade de sincronização durante o teste de regras e a execução de ações, visto que as regras de cada tarefa podem testar e modificar o estado de outras tarefas na mesma pool em segurança.

2.1.2 Verificação de modelos

Os sistemas distribuídos tolerantes a falhas estão propensos a erros de comunicação devido a falhas que afetam as interações de passagem de mensagens. Uma técnica usada para encontrar estes erros é a verificação de modelos [2], através da qual se explora o espaço de estados do sistema de forma automatizada e exaustiva. Neste técnica dado um modelo de estados finitos de um sistema e uma propriedade formal, é verificado sistematicamente se essa propriedade é válida nesse sistema.

Um modelo do sistema consiste num autómato composto por um conjunto de estados finitos e transições, no qual os estados contêm informação sobre os valores das variáveis e as transições descrevem a evolução do sistema de um estado para outro. Em sistemas realísticos, os autómatos de estados finitos são descritos usando uma linguagem de descrição de modelos, como por exemplo a linguagem de programação *Promela*, ou uma

linguagem de descrição de *hardware* como o *VHDL* ou *Verilog*.

A verificação de modelos permite especificar uma grande variedade de propriedades do sistema, tais como se o sistema faz o que é suposto fazer, ausência de pontos de bloqueio, propriedades de segurança ("algo mau nunca acontece") e propriedades de animação ("algo bom irá eventualmente acontecer"). Após a especificação, um verificador de modelos verifica a validade das propriedades em todos os estados do modelo do sistema, assegurando que este funciona corretamente.

Esta abordagem apresenta algumas desvantagem da explosão do espaço de estados, que consiste em ter um espaço de estados demasiado grande e portanto difícil de gerir, levando a que o tempo de exploração do estado seja igualmente grande.

2.2 Linguagens de protocolos

Nesta secção começamos por introduzir os tipos de sessão [4] para dar contexto ao nosso trabalho e às linguagens de descrição de protocolos que iremos apresentar neste capítulo. Depois iremos introduzir o *Scribble* [7], uma das primeiras linguagens de descrição de protocolos usada para especificar as interações de comunicação nos sistemas distribuídos, seguida de uma extensão [9] que permite ao *Scribble* especificar protocolos com interrup-ções. Por fim, apresentamos uma linguagem para especificar programas em *MPI* [11] e uma abordagem que controla a comunicação usando contratos de canais [5].

2.2.1 Tipos de sessão

Os tipos de sessão [4] são um formalismo para estruturar a interação e controlar os processos que participam na comunicação e o seu comportamento. Estes tipos permitem representar de forma abstrata a comunicação que passa nos canais de comunicação e detetar erros subtis na implementação de protocolos de comunicação.

A nível global existem os tipos de sessão globais, que são especificações abstratas de sistemas que dão uma vista de toda a comunicação do sistema. A nível local existem os tipos de sessão locais, que são protocolos que descrevem as interações de um participante específico, este participante corresponde a um processo *endpoint*. Os tipos de sessão globais são projetados em tipos de sessão locais e de seguida mostramos como ambos são estruturados.

Na figura 2.1 podemos ver algumas interações entre um cliente (User) e um multibanco (ATM). O cliente começa por enviar um identificador para o multibanco, que lhe responde com sucesso ou falha. Se responder sucesso o cliente pode pedir para fazer um depósito ou um levantamento, especificando em ambos os casos a quantia. O multibanco irá responder com o saldo no primeiro caso, no segundo caso dá o dinheiro ao cliente (dispense) ou diz que a quantia é superior ao saldo (overdraft) e termina a interação. Se no início o multibanco responder falha a interação termina imediatamente.

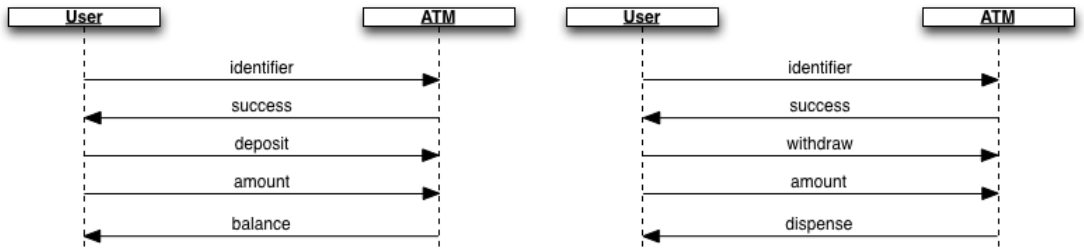

Figura 2.1: Diagramas de sequência para representar as interações entre o cliente e o multibanco [4].

Na figura 2.2 mostramos uma descrição global que representa estas interações. As interações têm a forma mostrada na linha 1, na qual os participantes envolvidos são identificados e separados por uma seta para indicar a direção da mensagem, para além disso ainda são indicados os dados da mensagem.

Caso existam várias opções de dados que podem ser enviados, estas são agrupadas usando chavetas e separadas pelo símbolo $[]$, como mostrado nas linhas 9-11. Por fim, é usada a palavra reservada end para terminar a interação.

A partir da descrição global é obtido um tipo global que representa a interação, para tal basta substituir os dados da mensagem pelos seus tipos. Por exemplo, na linha 1 o identificador é substituídos pelo tipo String.

```
1  User → ATM : identifier.
2  ATM → User :
3      { success : User → ATM :
4          { deposit :     User → ATM : amount.
5                          ATM → User : balance.
6                          end
7          [] withdraw :   User → ATM : amount.
8                          ATM → User :
9                              { dispense : end
10                             [] overdraft : end
11                             }
12         }
13     [] failure : end
14     }
```

Figura 2.2: Descrição global que representa a comunicação entre o cliente e o multibanco [4].

Depois através do tipo global é obtido um tipo de sessão local para cada participante, para tal basta remover as setas e usar os símbolos ! ou ? para identificar se o participante está a enviar ou a receber a mensagem, respetivamente. O símbolo $\oplus$ é usado para selecionar uma opção e $\&$ para quando o participante oferece uma escolha a outro participante. Na figura 2.3 é representado o tipo de sessão local para o cliente. O tipo de sessão local para o multibanco é semelhante ao do cliente, basta trocar os símbolos ! com ? e $\oplus$ com $\&$.

Por vezes tipos de sessão binários não são suficientemente poderosos para descrever e validar interações que envolvem mais do que duas partes, porque ao abstrair um protocolo como tipos de sessão separados iria gerar uma perda de informação de sequência. Para que

isso não aconteça o protocolo deve ser representado numa única sessão como um todo. Para tal em [8] a teoria dos tipos de sessão binários é estendida para suportar sessões assíncronas com múltiplos parceiros, na qual é introduzida a noção de tipos que lidam com interações que envolvem vários participantes.

```
1 ! String. &{ success : ⊕{ deposit : ! Real. ? Real. end
2                         [] withdraw : ! Real.
3                                       { dispense : end
4                                       [] overdraft : end
5                                       }
6                          }
7             [] failure : end
8             }
```

Figura 2.3: Tipo de sessão local que representa a interação do ponto de vista do cliente [4].

Nesta teoria estendida são utilizados canais de sessão para as interações entre vários participantes, um por cada par de *endpoints* que interagem e em cada interação o canal usado é identificado. Os tipos de sessão globais e locais em sessões com múltiplos parceiros são semelhantes aos usados nas sessões binárias, mas têm a adição do identificador do canal na sintaxe.

Quando um canal é usado em duas comunicações, para que a conversa nessa sessão proceda de forma correta, as ações de envio e receção devem ser ordenadas temporalmente. Se um tipo global satisfaz este princípio então específica um protocolo seguro, isto é, um protocolo em que as ações de envio/receção têm uma correspondência e portanto não ficam bloqueadas à espera de uma ação diferente, e pode ser uma base para garantir, através da verificação de tipos, que o comportamento dos processos é seguro. As ações de envio e receção são ordenadas, assim uma ação precede outra. Usando este esquema é possível ordenar as ações de comunicação estabelecendo relações causais nos tipos globais. Para induzir causalidade tem que se ter algumas condições em consideração, tais como se os canais usados nas operações são o mesmo ou não, se as operações são do mesmo tipo e quem são os participantes que estão a enviar e a receber mensagens. Dependendo destas condições podemos inferir se as operações têm uma relação de dependência ou não.

2.2.2 Scribble

O *Scribble* [7] é uma linguagem de descrição de protocolos que descreve num nível elevado as interações de comunicação, baseada na teoria dos tipos de sessão com múltiplos parceiros [5]. Os protocolos *Scribble* fornecem uma descrição global das interações entre todos os participantes, na qual cada participante tem um papel (*role*) específico e em cada operação de comunicação é identificado quem envia e quem recebe a mensagem. Este protocolo global é definido usando uma gramática especial, como a utilizada no protocolo exemplo na figura 2.4.

Um protocolo *Scribble* começa com um ou mais imports, que podem ser tipos de mensagens ou outros protocolos que serão utilizados na definição do protocolo. Na linha 1 temos um exemplo de um **import** do tipo de mensagem Message. Após os imports vem a definição do protocolo, que consiste na palavra reservada **protocol**, o nome e o corpo do protocolo.

O corpo do protocolo é composto por uma ou mais declarações de papéis e pela descrição da interação. Os papéis representam as *endpoints* participantes e são declarados como mostra a linha 4, na qual são declarados os papéis You e World. Após a declaração dos papéis vem a descrição da interação, que específica uma ou mais interações entre os papéis. Para definir estas interações é usada uma gramática que apresenta vários formatos de interações, alguns dos quais estão descritos na figura 2.5.

```
1 import Message;
2
3 protocol GreetWorld {
4     role You, World;
5     greet(Message) from You to World;
6 }
```

Figura 2.4: Protocolo global escrito através da linguagem *Scribble* [7].

O formato interação consiste no envio de uma mensagem de um determinado tipo. A mensagem pode ser um tipo básico, como um tipo primitivo de uma linguagem de programação (por exemplo o tipo **int** em *Java*), um tipo definido pelo utilizador ou um tipo composto, que contém um nome de operador aplicado a uma sequência de mensagens como *OpName(ValType1,..,ValTypeN)*.

O formato sequenciação representa uma série de interações ordenadas, onde duas interações que envolvem o mesmo papel podem ser ordenadas temporalmente, isto é, uma interação *I1* que é declarada antes de uma interação *I2*, e ambas envolvem o mesmo papel, então *I1* acontece antes de *I2*. Por outro lado temos o formato sem ordem que representa interações que podem acontecer em qualquer ordem.

O formato escolha dirigida representa uma escolha de interações, no qual um papel escolhe um tipo de mensagem e continua com a interação que segue esse cenário. Começa-se por identificar o papel que vai escolher e os outros papéis que vão participar nas interações, depois são definidos os cenários dentro de chavetas. Cada cenário é definido por um tipo de mensagem seguido de uma interação. Usando este formato o papel pode decidir qual será a sua próxima ação.

Por fim, o formato protocolo encaixada instancia uma nova conversa seguindo um protocolo específico. Começa com a palavra reservada **run**, seguida pela nome do protocolo e os argumentos necessários para o instanciar, como por exemplo os papéis que participam no protocolo. O *Scribble* tem mais formatos de interação, mas os apresentados já dão uma ideia do poder desta linguagem.

A especificação das interações começa pela definição do tipo de sessão global que corresponde a um protocolo *Scribble*. Depois o tipo global é projetado num papel es-

pecífico, originando um tipo de sessão local para cada papel. Através de um tipo global, estipula-se o conjunto de todas as regras de interação incluindo todos os participantes, por outro lado o tipo de sessão local permite saber quais as regras de comportamentos para um papel específico na conversa. A partir das regras de tipificação é derivado um algoritmo de tipificação que pode validar em tempo de compilação se o programa está de acordo com o tipo de sessão local projetado, assegurando desta forma que as conversas são livres de erros.

```
1. Interação
msgType from role1 to role2;

2. Sequenciação
I1; I2; ... In

3. Sem ordem
I1 & I2 & ... In

4. Escolha dirigida
choice from role1 to role2,.., roleK {
  msgType: I1
  ..
  msgType: In
}

5. Protocolo encaixado
run Protocol(param1,.., paramk, roleInChild1=roleInParent1,.., roleInChildn=
    roleInParentn);
```

Figura 2.5: Alguns construtores da linguagem *Scribble* [7].

2.2.3 Extensão do Scribble

Em [9] é apresentado uma extensão do *Scribble* que suporta conversas assíncronas com interrupções. Com esta extensão um fluxo de interações pode ser interrompido de forma assíncrona, como por exemplo, chamadas ao serviço com *timeout* e aplicações *publish-subscribe* nas quais o consumidor pode pedir para pausar e retomar os *feeds*.

Estas conversas com interrupções são descritas através dos protocolos *Scribble* com a adição de um construtor de interrupções. Este construtor contém um corpo principal com as ações do protocolo e um conjunto de assinaturas de mensagens de interrupção, estipulando que cada participante ou segue o protocolo especificado no corpo até terminar, ou levanta/deteta uma interrupção em qualquer altura durante o protocolo e sai do corpo principal.

Na figura 2.6 é apresentado um protocolo *Scribble* com interrupções que controla o acesso de utilizadores a sensores. Primeiro um cliente (User) envia uma mensagem request a um controlador (Controller) pedindo para usar o sensor durante um período de tempo, de seguida o controlador envia uma mensagem start a um agente (Agent).

Depois são declaradas duas interrupções encaixadas. Na interrupção exterior o fluxo pode ser interrompido por duas situações, quando o cliente não quer receber mais dados do sensor, então ele envia uma mensagem stop para o agente e o controlador. Ou quando o

controlador interrompe o fluxo ao enviar uma mensagem timeout para o cliente e o agente, porque o tempo da sessão expirou. Em ambas as situações o protocolo termina para todos os participantes.

O corpo da interrupção exterior é uma recursão com a etiqueta X, causada pela declaração na linha 18 que faz com que o fluxo do protocolo volte para o início da recursão, este ciclo continua indefinidamente até que uma das interrupções seja chamada.

Dentro da recursão X existe a interrupção interior para a opção em que o cliente pausa o fluxo, na qual tem lugar a recursão Y em que o agente envia repetidamente mensagens data ao cliente, que podem ser interrompidas pelo cliente com uma mensagem pause. A recursão Y é seguida pelo envio de uma mensagem resume do cliente para o agente antes do protocolo voltar ao topo da recursão X.

```
1  global protocol ResourceAccessControl (role User as U,
2          role Controller as C, role Agent as A) {
3      req(duration:int) from U to C;
4      start() from C to A;
5      interruptible {
6          rec X {
7              interruptible {
8                  rec Y {
9                      data() from A to U;
10                     continue Y;
11                 }
12             } with {
13                 pause() by U;
14             }
15             resume() from U to A;
16             continue X;
17         }
18     } with {
19         stop() by U;
20         timeout() by C;
21     }
22 }
```

Figura 2.6: Protocolo *Scribble* com interrupções [9].

O desenvolvimento de uma aplicação orientada à comunicação começa pela especificação das interações pretendidas na forma de um protocolo global, usando a linguagem de descrição de protocolos *Scribble*, como mostrado no exemplo acima. Depois uma ferramenta verifica se o protocolo global satisfaz certas propriedades de boa formação, tais como não haver ambiguidade entre os participantes sobre que cenário seguir e ausência de pontos de bloqueio entre fluxos paralelos. A partir do protocolo global válido a ferramenta gera os protocolos locais para cada participante, representando a comunicação do protocolo global da perspetiva de um participante.

Para assegurar a segurança global de um sistema na presença de interrupções assíncronas cada *endpoint* é monitorizado em tempo de execução, verificando se cada execução local está de acordo com o protocolo especificado. Para tal é usada uma API em *Python* para programar conversas com interrupções que fornece funcionalidades para início e adesão de sessão, *send/receive* básico e lidar com mensagens de interrupção.

Quando uma conversa é iniciada o monitor em cada *endpoint* gera uma máquina de

estados finitos (*FSM*) que representa ao comportamento a nível de comunicação daquele role. O monitor compara as ações de comunicação realizadas pelo *endpoint* e as mensagens que chegam de outros *endpoints* contra as transições permitidas na *FSM*.

Ao verificar localmente cada *endpoint*, o monitor assegura que o progresso global do sistema como um todo está de acordo com o protocolo global e que ações ilegais realizadas por um *endpoint* corrupto não conseguem corromper o estado do protocolo dos outros *endpoints*.

2.2.4 Protocolos para programas MPI

Em [11] é introduzida uma abordagem que consiste em especificar e verificar protocolos para programas *Message Passing Interface (MPI)* , que é baseada nos tipos de sessão com múltiplos parceiros. O *MPI* é uma especificação de biblioteca para envio e receção de mensagens, usada para programar aplicações paralelas de alto desempenho. O *MPI* oferece várias formas de comunicação coletiva para sincronização e *broadcasting* para cada membro do *cluster*. O *MPI* só está disponível para as linguagens de programação *C* e *Fortran*.

Num único programa *MPI* é especificado o comportamento de todos os processos, no qual cada processo trabalha em dados diferentes e são utilizadas chamadas a primitivas *MPI* sempre que os processos precisam de fazer trocas de dados. Cada processo é identificado por um *rank*, que permite que processos diferentes executem em paralelo e desempenhem ações distintas. Para além disso, oferece comunicação ponto-a-ponto, coletiva e *one-sided*.

Na comunicação ponto-a-ponto são utilizadas as primitivas *MPI_Send* e *MPI_Recv*, que representam as operações básicas de envio e receção, respetivamente. O *MPI* oferece várias primitivas para comunicação coletiva, como o *MPI_Bcast* que faz o *broadcast* de dados para outros processos e o *MPI_Gather* que junta dados de diferentes processos numa única estrutura.

Para sincronizar um grupo de processos é utilizada a primitiva *MPI_Barrier*, quando um processo chama esta primitiva fica bloqueado até que todos os processos do grupo também tenham chamado a primitiva, originando um ponto de sincronização.

Nesta abordagem um protocolo especifica todas as interações entre os participantes de um ponto de vista global. Para descrever os protocolos é usada uma linguagem de tipos, como é mostrado na figura 2.7. Na linha 2 é identificado o número de participantes, depois o participante com *rank* 0 faz broadcast para todos os participantes da dimensão do problema, seguido por uma operação scatter que divide um array de tamanho n por todos os participantes. Nas linhas 7-8 são trocadas mensagens ponto-a-ponto, seguidas por um allreduce em que cada participante comunica um número, depois é calculado o máximo entre esse números e é distribuído por todos os participantes. Por fim, na linha 13 o participante com *rank* 0 junta num array dados provenientes de todos os participantes.

Para verificar se o programa *MPI* está de acordo com o protocolo é usada a ferramenta *VCC* [3]. Para que o *VCC* consiga perceber o programa é necessário adicionar anotações no código fonte e substituir o que o *VCC* não suporta. Depois é verificado se o programa está bem formado. Caso o programa esteja bem formado é gerado um cabeçalho na linguagem *C* que descreve o protocolo no formato *VCC*.

```
1  protocol fdiff {
2      size p: {x: positive | x > 1};
3      broadcast 0 n: {x: natural | x % p = 0};
4      scatter 0 float[n];
5      loop {
6          foreach i: 0 .. p - 1 {
7              message i, (p + i -1) % p float;
8              message i, (i + 1) % p float
9          };
10         allreduce max float
11     };
12     choice
13         gather 0 float [n]
14     or
15         {}
16 }
```

Figura 2.7: Protocolo que especifica um programa *MPI* [11].

O próximo passo é incluir marcas simples no código fonte que controlam aspetos de fluxo, como ciclos coletivos/escolhas, que são difíceis de inferir automaticamente usando técnicas padrão de análise estática. Depois é escrita uma versão em contracto anotado das primitivas *MPI*, que assegura a validade do programa de acordo com o protocolo, cada contrato descreve uma primitiva *MPI* com um conjunto de pré e pós-condições.

Neste ponto, uma ferramenta de anotações automática gera as anotações necessárias, guiada pelas chamadas às primitivas *MPI* e pelas marcas que são expandidas para anotações mais complexas. Na maioria dos casos, os programas necessitam da introdução manual de anotações complementares. Após este passo o processo de verificação pode ser iniciado.

2.2.5 Protocolos de canais

No *Singularity OS* [5] a maioria do código é escrito em *Sing#*, que é uma extensão do *C#*, e os processos não partilham memória, portanto só comunicam através de mensagens. A comunicação é feita por canais bi-direcionais, que são canais sem perdas, que entregam as mensagens ordenadas e recebem-nas na mesma ordem pela qual foram enviadas. Cada canal consiste em dois *endpoints*, um denominado *import* (Imp) e o outro *export* (Exp), que são distinguidos pelos tipos *C.Imp* e *C.Exp*, onde *C* é o contrato do canal que controla a interação.

Os processos no *Singularity* mantêm pilhas independentes e para enviar dados de um processo para outro é usada um pilha de troca, que apenas contém dados que podem ser movidos entre processos. Os processos têm apontadores para a sua pilha e para a pilha de troca, mas a pilha de troca apenas aponta para si própria. Cada bloco de memória

na pilha de troca apenas pertence a um processo em qualquer altura da execução, mas é possível que os processos tenham apontadores para blocos na pilha de troca que não lhes pertencem. Para impedir que os processos acedam à memória por referências que não lhes pertencem os blocos são verificados estaticamente de acordo com as regras de posse de blocos.

Os contratos de canais em *Sing#* descrevem as mensagens trocadas, os tipos dos argumentos das mensagens, a direção das mensagens e o conjunto de estados do protocolo. Em cada estado são declaradas as possíveis mensagens e os estados a que estas levam. Na figura 2.8 é apresentado um contrato de canal para *drivers* de dispositivos de rede. O contracto começa com a palavra reservada contract e o nome do contrato, seguido do corpo do contrato, que consiste em todos os estados possíveis. No início do corpo do contrato são declaradas todas as mensagens envolvidas nas interações e pode-se definir a direção usando as palavras reservadas in (Exp para Imp) e out (Imp para Exp), como mostrado nas linhas 12-13.

```
1  contract NicDevice {
2      out message DeviceInfo (...);
3      in message RegisterForEvents(NicEvents.Exp:READY evchan);
4      in message SetParameters (...);
5      out message InvalidParameters (...);
6      out message Success();
7      in message StartIO();
8      in message ConfigureIO();
9      in message PacketForReceive(byte[] in ExHeap pkt);
10     out message BadPacketSize(byte[] in ExHeap pkt, int mtu);
11     in message GetReceivedPacket();
12     out message ReceivedPacket(NetworkPacket! in ExHeap pkt);
13     out message NoPacket();
14
15     state START: one {
16         DeviceInfo! → IO_CONFIGURE_BEGIN;
17     }
18     state IO_CONFIGURE_BEGIN: one {
19         RegisterForEvents? →
20         SetParameters? → IO_CONFIGURE_ACK;
21     }
22     state IO_CONFIGURE_ACK: one {
23         InvalidParameters! → IO_CONFIGURE_BEGIN;
24         Success! → IO_CONFIGURED;
25     }
26     state IO_CONFIGURED: one {
27         StartIO? → IO_RUNNING;
28         ConfigureIO? → IO_CONFIGURE_BEGIN;
29     }
30     state IO_RUNNING: one {
31         PacketForReceive? → (Success! or BadPacketSize!)
32             → IO_RUNNING;
33         GetReceivedPacket? → (ReceivedPacket! or NoPacket!)
34             → IO_RUNNING;
35     ...
36     }
37 }
```

Figura 2.8: Um contrato de canal para *drivers* de dispositivos de rede [5].

Depois os estados são definidos como no exemplo das linhas 15-17, neste caso o estado é identificado pelo nome START e dentro das chavetas é definida a sequência de mensagens. O estado START consiste no envio de uma mensagem DeviceInfo, após o en-

vio a conversa fica no estado IO_CONFIGURE_BEGIN. Os símbolos ! e ? são símbolos de direção, usa-se ! para a direção de Exp para Imp e ? de Imp para Exp. A seta é usada para definir o que acontece a seguir, geralmente é seguida pelo nome de um estado ou a troca de outra mensagem. No contrato é possível criar outros canais controlados por outros contratos, como na linha 3 é definida a mensagem RegisterForEvents que tem um argumento especial do tipo NicEvents.Exp:READY, que cria um segundo canal controlado pelo protocolo NicEvents quando essa mensagem é recebida no estado IO_CONFIGURE_BEGIN.

As regras para lidar com os blocos da pilha de troca permitem que um compilador verifique estaticamente se os processos apenas acedem à memória que lhes pertence e se as operações de envio/receção nos canais nunca são aplicadas no estado errado. Estas regras dizem respeito à posse de apontadores para a pilha de troca que é transferida de processo para processo aquando da verificação.

O uso de especificações nos canais de comunicação ajuda a detetar erros dos programadores cedo, nomeadamente em tempo de compilação, reduzindo a dificuldade do modelo de programação baseado em mensagens.

2.3 Considerações finais

Neste capítulo foram apresentadas várias linguagens para representar a comunicação dos sistemas distribuídos, primeiro apresentámos os tipos de sessão que servem como base para a maioria das linguagens de protocolos, o *Scribble* que especifica sistemas distribuídos. Uma extensão para o *Scribble* que suporta conversas assíncronas com interrupções. Uma linguagem que permite especificar e verificar protocolos para programas *MPI*. Os contratos de canais que controlam a comunicação dos canais e são escritos usando a linguagem *Sing#*. No entanto nenhuma das linguagens apresentadas permite tratar falhas, que é o objetivo do nosso projeto.

Capítulo 3

Uma linguagem de protocolos para descrever sistemas distribuídos tolerantes a falhas

O nosso projeto consiste na especificação de sistemas distribuídos tolerantes a falhas, de forma a facilitar a sua implementação. Para tal criámos uma linguagem para especificar a comunicação nesses sistemas, baseada na teoria dos tipos de sessão com múltiplos parceiros [8]. Esta linguagem permite a definição de protocolos globais, que oferecem uma vista geral da comunicação, que depois são projetados em protocolos locais, que mostram a comunicação pela perspetiva de cada um dos processos participantes na comunicação.

Neste capítulo começamos por fornecer uma vista geral do nosso projeto. De seguida introduzimos os protocolos e as regras de tradução que permitem gerar protocolos locais a partir de protocolos globais. Para ilustrar estes conceitos usamos o protocolo de confirmação em duas fases (*Two-Phase Commit* - 2PC), para o qual criámos protocolos globais e locais usando a nossa linguagem.

3.1 Visão geral

Na figura 3.1 está representada a visão geral do nosso trabalho. Começamos por construir um protocolo global através da linguagem definida na secção 3.2. Este protocolo especifica todas as interações de comunicação entre todos os participantes, representando um esquema global da comunicação. As interações de comunicação podem ser especificadas em apenas um protocolo global que contém toda a comunicação ou podem ser divididas em vários protocolos, separando a comunicação que envolve apenas um subconjunto dos participantes. Na secção 3.3 usamos o exemplo 2PC para explicar em detalhe como é feita esta divisão de protocolos.

Um protocolo global consiste em várias trocas de mensagens entre os diversos processos participantes na comunicação, sendo que em cada troca é identificado quem envia a mensagem e quem a recebe, assim como o tipo de mensagem que é trocada. Na secção

3.3 explicamos todos os detalhes da construção de protocolos globais. Com o protocolo global formado este é projetado em protocolos locais, um para cada um dos participantes envolvidos na comunicação, usando as regras de tradução apresentadas na secção 3.4.

Um protocolo local representa a perspetiva que um participante tem da comunicação, por isso só inclui as interações que envolvem esse participante específico. Tal como no protocolo global, o protocolo local também consiste em várias trocas de mensagens, sendo que em cada troca apenas é identificado o participante com quem se está a comunicar.

Cada processo tem um programa individual, escrito numa linguagem de programação específica, que implementa as suas ações. Este programa tem que estar de acordo com o protocolo local do participante, para tal introduzimos na secção 5 um modelo que permite escrever a parte da comunicação do programa, para que este esteja em conformidade com as trocas de mensagens especificadas no protocolo local.

Através do modelo de programação podemos identificar facilmente que partes do programa correspondem a cada troca representada no protocolo local de um participante. No futuro poder-se-á criar um verificador que usando protocolos locais verifica se o programa está em conformidade com o protocolo local e portanto não irá gerar erros de comunicação durante a execução. Mas neste projeto não criamos tal verificador, nem é um dos nossos objetivos.

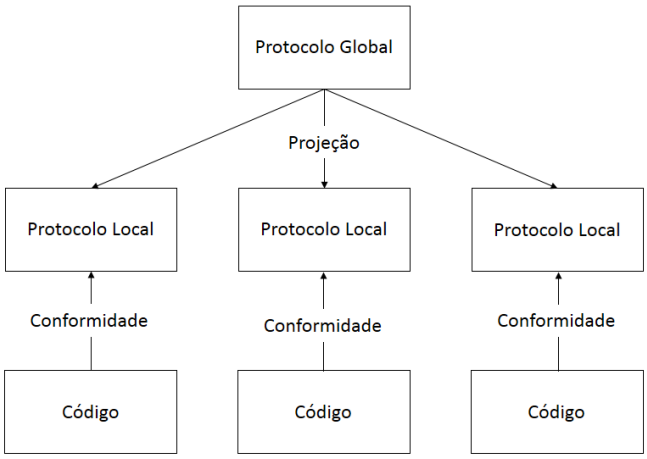

Figura 3.1: Visão geral.

3.2 Linguagem de especificação de protocolos

Para construir os protocolos criámos uma linguagem para protocolos globais e locais, que permite representar as interações de comunicação entre os vários participantes. Esta sintaxe é baseada nos tipos de sessão referidos na secção 2.2.1 e noutras linguagens de protocolos referidas na secção 2.

3.2.1 Linguagem dos protocolos globais

A sintaxe dos protocolos globais é apresentada na figura 3.2. A sintaxe dos protocolos locais é semelhante, sendo a diferença mais evidente a forma como a troca de mensagens é representada: no protocolo global existe uma única palavra reservada para representar uma troca de mensagens, que representa o envio e a receção da mensagem, no protocolo local representamos apenas um dos lados da troca de mensagens, usamos palavras reservadas especificas para representar o envio ou a receção da mensagem. Existem palavras reservadas diferentes para representar um envio ou uma receção de um para um ou entre um e múltiplos participantes.

```
Base sets
    label, l
    variables in expressions, x
    role identifiers, rid
    role names, role
    protocol names, pid
    parameters names, param

Shared(expressions, propositions, role id arity)
    e::= ...|−1|0|1|...
         e+e|e∗e|...
         x
    p::= true|p∧p|p→p|...
         e>e|e=e|...
    A::= [e|p]
    S::= (rid: role A,.., rid: role A)

Global protocols
    G::= global protocol pid S M
    M::= message rid→rid {e?:B,..,e?:B,e?:T?}|
         exists rid:role.M|
         M + M|
         pid(rid,..,rid)|
         end
    B::= l(param,..,param)?: M
    T::= timeout(rid:role A)?: M
```

Figura 3.2: Sintaxe da linguagem de protocolos globais.

No topo da figura introduzimos um conjunto identificadores usados para representar alguns nomes na sintaxe de ambos os protocolos. Cada mensagem é identificada por uma etiqueta, que permite que o recetor perceba que ação deve tomar consoante o tipo da mensagem que recebe. Um exemplo de uma etiqueta é vote_request , que representa um pedido de voto. Para representar as etiquetas das mensagens é utilizada o identificador l. Quando a etiqueta não é suficiente para representar toda a informação que o emissor tem que enviar, essa informação extra é passada nos parâmetros da mensagem, que são representados pelo identificador param.

Por vezes é necessário representar informação que pode variar, como por exemplo o número de processos que executam um programa, essa informação é definida geralmente em tempo de execução. A esta informação variável chamamos variáveis e na sintaxe são representadas pelo identificador x. Para representar os participantes na comunicação é utilizado o identificador rid, que pode representar um único participante ou um conjunto

de participantes que se comportam todos do mesmo modo. Cada participante desempenha um papel específico na ação, representado na sintaxe pelo identificador role. Em sistemas distribuídos geralmente existem os processos clientes e os processos servidores, o cliente pede alguma informação ao servidor e este responde ao pedido, neste caso consideramos que existem dois tipos de papéis, o cliente e o servidor. Por fim, o nome do protocolo é representado pela identificador pid.

De seguida são apresentadas algumas partes da sintaxe que são partilhadas pelos protocolos globais e locais. As expressões, representadas pelo símbolo não terminal e, podem ser qualquer número inteiro ou uma variável. Para criar expressões mais complexas podem-se aplicar operações matemáticas, como a soma, a multiplicação, etc, entre expressões para combinar variáveis e números, como por exemplo para representar um número par de participantes pode-se usar a expressão 2∗x.

As proposições, representadas pelo símbolo não terminal p, fornecem informação sobre variáveis em expressões. Podem limitar os valores de uma variável através dos símbolos maior (>), menor (<) ou igual (=). Por exemplo pode-se limitar os valores da variável x, mencionada no exemplo acima, para que esta seja pelo menos um. Para tal usando a proposição x>=1 em conjunto com a expressão 2∗x indicamos que o número de participantes é par e positivo.

O símbolo não terminal A representa a aridade dos participantes, ou seja, quantos participantes que fazem parte de um conjunto de participantes. Para definir a aridade usamos uma expressão seguida de uma proposição. Recorrendo novamente ao exemplo acima, a aridade do conjunto de participantes pode ser definida como [2∗x|x>=1].

Por fim, o símbolo não terminal S representa a assinatura do protocolo, na qual são identificados os participantes envolvidos na comunicação. Cada participante ou conjunto de participantes é declarado usando um identificador único, seguido do seu papel e a aridade. Os identificadores definidos na assinatura são usados nas trocas de mensagens do protocolo para identificar quem envia e recebe a mensagem. Através do papel sabemos qual a ação que o participantes desempenha. A aridade é opcional, quando declaramos apenas um participante esta é omitida, quando se trata de um conjunto de participantes é necessário definir quantos participantes fazem parte desse conjunto.

Para construir os protocolos globais começa-se por declarar o cabeçalho do protocolo, usando as palavras reservadas global protocol que identificam o protocolo como sendo um protocolo global, de seguida declaram-se o nome, a assinatura e o corpo do protocolo. O cabeçalho do protocolo é inspirado na linguagem *Scribble*, descrita nas secções 2.2.2 e 2.2.3.

O corpo do protocolo consiste em múltiplas trocas de mensagens que representam as várias operações de comunicação que ocorrem entre os participantes. As trocas de mensagens apresentam vários formatos, representados na sintaxe pelo símbolo não terminal M. O formato mais simples de uma troca de mensagens consiste na palavra reservada

message seguida por dois identificadores de participantes separados por uma seta. A seta representa a direção da mensagem, assim o participante que envia a mensagem está antes da seta e o que recebe encontra-se após a seta. De seguida é definida uma lista com as opções de mensagens que podem ser trocadas.

Cada opção de mensagem é representada pelo símbolo não terminal B e é precedida por uma expressão que identifica o número de mensagens que o recetor necessita de receber para avançar na comunicação. Esta expressão é opcional, tal como representado na sintaxe pelo símbolo de interrogação acima do símbolo mão terminal e. Para que o participante que espera pela mensagem não fique bloqueado para sempre à espera, é usado um temporizador que quando atinge um determinado limite de tempo sem receber uma mensagem que se enquadre nas opções avança para outro passo na comunicação. A este caso damos o nome de *timeout* e é representado no fim da lista de opções pelo símbolo não terminal T. Quando existe um conjunto de participantes a receber mensagens, o *timeout* pode ser precedido por uma expressão opcional que indica quantos participantes têm que fazer *timeout* para avançar para o próximo passo.

Por vezes torna-se necessário criar um novo participante para além dos participantes declarados na assinatura do protocolo. Por exemplo, pode ser necessário identificar um participante que pertence a um conjunto para representar uma troca de mensagens que o envolva apenas a ele e não todo o conjunto. Para tal é utilizada a palavra reservada exists , seguida do identificador e papel do participante. De seguida é declarada a troca de mensagens que envolve esse participante. A palavra reservada exists significa que existe um participante com identificador rid e com papel role que envia ou recebe uma mensagem de outro participante.

Após uma troca de mensagens podem existir várias alternativas para continuar a comunicação, para representar este caso as trocas de mensagens são separadas pelo símbolo + indicando que a comunicação pode seguir por qualquer uma delas.

Quando existem vários protocolos globais pode ser necessário invocar outro protocolo global dentro de um protocolo global para continuar a comunicação. Para tal basta declarar o seu nome e os identificadores dos participantes que fazem parte dele.

Por fim, para terminar a comunicação é usada a palavra reservada end e nesse ponto o protocolo termina para todos os participantes.

Cada opção de mensagem, representada pelo símbolo não terminal B, consiste na etiqueta da mensagem seguida pelos parâmetros da mensagem. Os parâmetros são opcionais, no caso em que estes não existem a opção consiste apenas na etiqueta seguida pela abertura e fecho de parênteses. No fim de cada opção de mensagem é declarada a próxima operação de comunicação.

O *timeout*, representado pelo símbolo não terminal T, consiste na palavra reservada timeout seguida da declaração de um participante. Esta declaração é opcional e serve para identificar o participante ou conjunto de participantes que fizeram *timeout*, quando

existem vários recetores. O participante é declarado tal como na assinatura do protocolo. Após declarar o *timeout* é especificada uma nova operação de comunicação.

3.2.2 Linguagem dos protocolos locais

A sintaxe para os protocolos locais, apresentada na figura 3.3 é bastante semelhante à dos protocolos globais. A diferença mais evidente é que nos protocolos globais existe apenas a troca de mensagens base, identificada pela palavra reservada message e nos protocolos locais existem vários tipos de envio/receção de mensagens.

```
Base sets
    label, l
    variables in expressions, x
    role identifiers, rid
    role names, role
    protocol names, pid
    parameters names, param

Shared(expressions, propositions, role id arity)
    e ::= ...| −1|0|1|...
          e+e|e∗e|...
          x
    p ::= true|p∧p|p→p|...
          e>e|e=e|...
    A ::= [e|p]
    S ::= (rid: role A,.., rid: role A)

Local protocols
    L ::= local protocol pid S N
    N ::= send rid {C,..,C}|
          multisend rid {C,..,C}|
          receive rid {C,..,C,O?}|
          multireceive rid {e?:C,..,e?:C,e?:O?}|
          exists rid:role.N|
          N + N|
          pid(rid,..,rid)|
          end
    C ::= l(param,..,param)?: N
    O ::= timeout(rid:role)?: N
```

Figura 3.3: Sintaxe da linguagem de protocolos locais.

Um protocolo local inicia-se com as palavras reservadas local protocol, que o identificam como sendo um protocolo local, seguidas do nome, assinatura e corpo do protocolo. O corpo do protocolo consiste em várias trocas de mensagens entre o participante a quem o protocolo local pertence e outros participantes. Na figura 3.4 apresentamos as quatro formas de envio/receção de mensagens na sintaxe de protocolos locais.

Existem duas formas de envio de mensagens, o send que consiste no envio de uma mensagem de um participante para outro participante, e o multisend que corresponde ao envio de mensagens de um participante para um conjunto de participantes, neste caso são enviadas tantas mensagens quanto a aridade do conjunto recetor. Ambas as formas de envio apresentam o mesmo formato, começam com a palavra reservada inicial, send ou multisend, seguida do identificador do recetor e a lista de opções de mensagens que podem ser enviadas. Como estas formas são de envio de mensagens não são utilizadas expressões

antes das opções de mensagens, nem existe a opção timeout, pois estas são características da receção de mensagens.

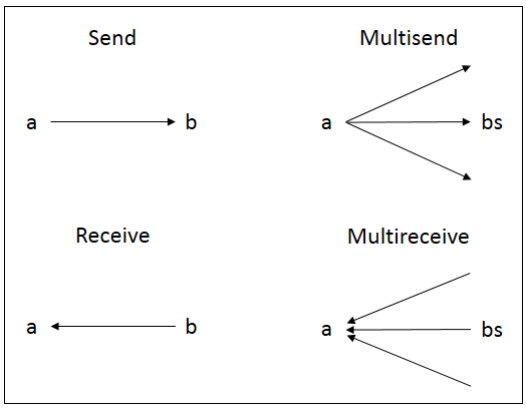

Figura 3.4: Formas de envio e recepção de mensagens.

Quanto às formas de receção de mensagens, existe o receive que consiste na receção de uma mensagem enviada por um participante para outro participante, e o multireceive que consiste na receção de múltiplas mensagens enviadas por um conjunto de participantes para um participante.

As formas de receção apresentam o mesmo formato que as formas de envio. Como se trata da receção de mensagens no fim da lista de opções aparece opcionalmente o timeout, representado pelo símbolo não terminal O, para impedir que o participante fique bloqueado à espera de mensagens. No multireceive as opções de mensagens podem ser antecedidas por uma expressão, que identifica o número de mensagens de cada opção que são necessárias receber para que o participante passe para a operação de comunicação seguinte. Estas expressões são opcionais, caso a expressão não seja definida espera-se por tantas mensagens quanto a aridade do conjunto de participantes.

Na sintaxe dos protocolos locais também existe a opção de criar um novo participante. Tal como na sintaxe dos protocolos globais para criar este novo participante é utilizada a palavra reservada exists , seguida do identificador e papel do participante. Após criar o novo participante é declarada a troca de mensagens em que este está envolvido.

Nos protocolos locais também existem várias alternativas de trocas de mensagens pelas quais se pode continuar a comunicação. Tal como na sintaxe dos protocolos globais as alternativas são separadas pelo símbolo +.

Para invocar outro protocolo onde se irá continuar a comunicação utiliza-se o nome do protocolo e os identificadores dos participantes que fazem parte dele.

Por fim, para terminar a comunicação é usada a palavra reservada end e nesse ponto o protocolo termina.

As opções de mensagens, identificadas na sintaxe pelo símbolo não terminal C, apresentam o mesmo formato que nos protocolos globais, consistem na etiqueta da mensagem

e os parâmetros são opcionais, tal como indicado pelo ponto de interrogação na sintaxe. No fim de cada opção de mensagem é declarada a próxima operação de comunicação.

O *timeout* é representado na sintaxe dos protocolos locais pelo símbolo não terminal O e consiste na palavra reservada timeout seguida da declaração opcional de um participante entre parêntesis. Num protocolo local apenas o participante a quem o protocolo pertence é que pode fazer *timeout*, mas pode ser necessário identificá-lo caso ainda não tenha um identificador individual. Para declarar o participante basta usar o identificador e papel do participante, separado por dois pontos. A aridade não é utilizada pois apenas se está a identificar um participante. Após declarar o *timeout* é especificada uma nova operação de comunicação.

3.3 Protocolos globais

Um protocolo global representa uma vista global da comunicação de um sistema distribuído, na qual todos os participantes estão incluídos e engloba todas as mensagens que podem ser trocadas entre eles. À semelhança dos protocolos de canais, apresentados na secção 2.2.5, um protocolo global apresenta vários estados, sendo que cada estado consiste numa troca de mensagens. Durante o desenvolvimento dos protocolos globais optámos por representar as trocas de mensagens por estado, porque facilita a leitura dos protocolos e torna-os mais simples. Cada estado é identificado pelo seu nome e inclui os identificadores do emissor e recetor, as opções de mensagens que podem ser trocadas nesse estado e por fim o nome do próximo estado em que continua a comunicação ou a palavra reservada de fim de protocolo.

De forma a separar partes da comunicação em que só alguns participantes estão incluídos são usados vários protocolos globais que representam partes separadas da comunicação. Os protocolos globais podem ser desencadeados a qualquer altura durante a comunicação. Um protocolo global pode ser desencadeado pela receção de uma mensagem ou um estado de um protocolo global pode invocar outro protocolo onde a comunicação irá continuar.

Cada protocolo global representa apenas uma execução do programa. Geralmente os sistemas distribuídos funcionam como cliente e servidor, sendo que um cliente faz um pedido a um servidor e o servidor atende vários pedidos de diversos clientes. Desta forma uma execução do protocolo global consiste na comunicação desde que o cliente faz o pedido até receber a resposta, o pedido seguinte já corresponde a outra execução.

3.3.1 Descrição do protocolo Two-Phase Commit

Para ilustrar a construção de um protocolo global usamos o protocolo 2PC [6], criado por Gray em 1978. Este protocolo é utilizado em bases de dados para completar transações que modificam dados em múltiplos servidores. No 2PC existem vários processos, cada

um a executar numa máquina diferente, que participam numa transação e têm que decidir em conjunto se vão fazer *commit*, isto é, se completam a transação localmente ou se a abortam.

No 2PC existem dois tipos de processos, um dos processos é chamado coordenador e é responsável por pedir os votos, juntá-los e informar os outros protocolos da decisão final, os restantes processos são chamados participantes e apenas têm que responder com o seu voto, ou seja, se decidem fazer *commit* ou abortar a transição.

Numa execução em que não existem falhas o protocolo consiste em duas fases, a fase de votação e a fase de decisão. As trocas de mensagens realizadas durante ambas as fases estão representadas na figura 3.5.

Na fase de votação o coordenador envia uma mensagem vote_request para todos os participantes. Quando um participante recebe uma mensagem vote_request , envia uma mensagem vote_commit para o coordenador avisando que está pronto para fazer o *commit* local da sua parte da transação, caso não esteja pronto envia uma mensagem vote_abort .

Na fase de decisão o coordenador junta todos os votos dos participantes. Se todos os votos são para fazer *commit* da transação, então o coordenador concorda com essa decisão e envia uma mensagem global_commit para os participantes. No entanto, se um participante votar para abortar a transação, o coordenador também decide abortar a transação e envia uma mensagem global_abort para os participantes, informando-os da decisão.

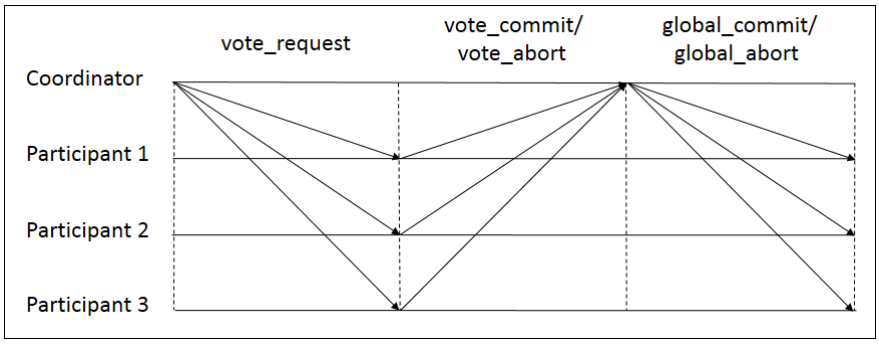

Figura 3.5: Trocas de mensagens no 2PC.

No entanto num sistema em que podem ocorrer falhas surgem alguns problemas. Tanto o coordenador como os participantes podem ficar parados à espera de mensagens, pois se um processo falhar os outros que ficam à espera de uma mensagem vinda dele não conseguirão avançar. Para resolver este problema são implementados mecanismos de *timeout*, ou seja, um processo que fica à espera de uma mensagem inicia um temporizador e define um determinado limite tempo, quando atinge esse limite não espera mais e realiza outra ação.

Neste protocolo existem três casos em que o coordenador ou um participante ficam bloqueados. O primeiro caso acontece quando um participante espera pela mensagem vote_request do coordenador. Se essa mensagem não for recebida ao fim de algum tempo

o participante aborta a transação localmente e envia uma mensagem vote_abort para o coordenador. Outro caso acontece quando o coordenador fica bloqueado à espera dos votos de cada participante. Se não receber todos os votos ao fim de algum tempo, o coordenador decide abortar a transação e envia uma mensagem global_abort para todos os participantes. Por último um participante pode ficar bloqueado à espera da decisão global enviada pelo coordenador. Se essa mensagem não for recebida ao fim de algum tempo, o participante não conseguirá decidir sozinho se deve abortar ou fazer *commit* da transação.

Uma solução para este caso consiste em contactar outro participante para o ajudar a saber qual foi a decisão global. Assim um participante P contacta um participante Q, enviando uma mensagem decision_request para pedir a decisão.

Se o participante Q recebeu a decisão global do coordenador envia uma mensagem dec_global_commit ou dec_global_abort para P, de acordo com a decisão que recebeu. Existe o caso em que Q ainda se encontra à espera da mensagem vote_request do coordenador, isto acontece quando o coordenador falha enquanto está a fazer o envio das mensagens vote_request , por isso o participante P recebeu a mensagem e Q não. Nesta situação é seguro abortar a transação, portanto Q envia uma mensagem dec_global_abort para P.

Por fim, ainda existe a situação em que Q também se encontra à espera da decisão do coordenador. Neste caso Q envia uma mensgem no_decision para o participante P, informando que não consegue decidir. Para contornar este problema, P tenta contactar outro participante para o ajudar a decidir. Caso todos os participantes estejam à espera da decisão, ninguém consegue decidir e ficam bloqueados até que o coordenador recupere. Quando o participante P está à espera da mensagem de Q e não a recebe ao fim de algum tempo, ocorre um *timeout* e P irá pedir a decisão a outro participante.

3.3.2 Protocolo global do Two-Phase Commit

Protocolo global da execução normal. O protocolo global apresentado na figura 3.6 representa a comunicação principal do 2PC entre o coordenador e os participantes. Como referido anteriormente existem dois tipos de processos, por isso no início do protocolo começa-se por declarar os papéis existentes. Para tal basta usar a palavra reservada **role** seguida do nome dos papéis, neste caso os papéis declarados são Coordinator e Participant .

De seguida escreve-se o cabeçalho como descrito na sintaxe de protocolos globais, na secção 3.2. O cabeçalho consiste nas palavras reservadas **global protocol**, o nome do protocolo, que neste exemplo é TwoPhaseCommit, e a assinatura do protocolo.

Na assinatura deste protocolo são identificados dois participantes, o coordenador e o conjunto de participantes. Os participantes são declarados usando um identificador único, o papel e a aridade. O coordenador é representado pelo identificador c e o seu papel é Coordinator, como é apenas um participante a aridade é omitida. Quanto ao conjunto de participantes é identificado por ps e o seu papel é Participant . Para definir a aridade de ps é utilizada a variável n para representar o número de participantes que fazem parte do

conjunto. Usa-se uma variável e não um número pois o número de participantes pode variar de execução para execução. Como tem que existir pelo menos um participante, usamos uma proposição para restringir os valores de n para este seja maior ou igual a um, ou seja, n>=1. Juntando a expressão e a proposição a aridade é definida como [n|n>=1].

```
1  role Coordinator, Participant
2
3  global protocol TwoPhaseCommit(c: Coordinator, ps: Participant [n|n>=1]){
4      PhaseOne:
5          message c → ps {
6              vote_request(): WaitReady
7              timeout(): WaitReady
8          }
9
10     WaitReady:
11         message ps → c{
12             n: vote_commit(): PhaseTwo
13             1: vote_abort(): PhaseTwo
14             timeout(): PhaseTwo
15         }
16
17     PhaseTwo:
18         message c → ps{
19             global_commit(): end
20             global_abort(): end
21             timeout(p:Participant): Decision(p).AskDecision
22         }
23 }
```

Figura 3.6: Protocolo global da execução normal.

De seguida é declarado o corpo do protocolo, que consiste em todos os estados que representam a comunicação realizada durante a execução normal, incluindo os *timeouts*. No protocolo TwoPhaseCommit existem 3 estados.

No primeiro estado, ao qual foi dado o nome PhaseOne, é representado o pedido de voto do coordenador aos participantes. Para definir um estado começa-se por indicar o seu nome seguido de dois pontos, tal como mostrado na linha 4. De seguida, nas linhas 5-8, é especificada a troca de mensagens. A troca de mensagens começa com a palavra reservada **message** seguida do identificador do emissor, a seta de direção de mensagens e o identificador do recetor. Como nesta troca é o coordenador que envia as mensagens para os participantes, o identificador c aparece antes da seta de direção e o identificador ps aparece após a seta. Depois do identificador do emissor abre-se chavetas e nas linhas seguintes são listadas as opções de mensagens que podem ser trocadas e a opção **timeout** caso seja aplicável a esta troca.

Como nesta troca de mensagens só pode ser enviada a mensagem vote_request, só é listada uma opção de mensagem. Para definir a opção usa-se a etiqueta seguida dos parâmetros da mensagem entre parênteses. Como neste caso não existem parâmetros os parênteses ficam vazios. No fim da opção é identificado o nome do estado onde a comunicação vai continuar se esta opção for selecionada. Como referido na descrição do protocolo após receber o vote_request os participantes enviam o seu voto para o coordenador, por isso a comunicação tem que continuar no estado que representa essa troca, que neste caso é o estado WaitReady.

Na linha seguinte é definida a opção **timeout**, usando a palavra reservada **timeout** seguida de parênteses tal como na opção de mensagens. Dentro dos parênteses pode ser identificado o participante que fez *timeout* quando desencadeia uma troca especifica entre este participante e outro, mas como neste caso não é necessário identificar o participante os parênteses ficam vazios. No fim desta opção é identificado o nome do estado onde a comunicação vai continuar quando ocorre um *timeout*. No protocolo 2PC se ocorrer um *timeout* o participante envia uma mensagem vote_abort para o coordenador, portanto após a opção **timeout** o participante segue para o estado WaitReady que representa essa troca.

No estado seguinte, cujo nome é WaitReady, é representado o envio das mensagens de voto dos participantes para o coordenador. Este estado é semelhante ao anterior, mas nesta troca de mensagens o emissor é o conjunto de participantes ps e o participante c é o recetor.

De seguida é definida a lista de opções de mensagens. Como referido no protocolo 2PC o coordenador espera pelas mensagens de voto de todos os participantes, se todas as mensagens recebidas têm a etiqueta vote_commit a comunicação avança para o próximo passo, mas basta receber uma mensagem vote_abort para que a comunicação avance. Para representar quantas mensagens de cada tipo o coordenador espera receber antes da comunicação avançar para o próximo estado são usadas expressões antes da etiqueta de cada opção. Assim a primeira opção começa com a variável n, que representa a aridade do conjunto de participantes, assim indica que o coordenador espera receber uma mensagem de cada um dos participantes. Após a variável n aparece a etiqueta vote_commit seguida do nome do estado onde a comunicação continua. No protocolo 2PC quando o coordenador recebe só votos para fazer *commit* envia uma mensagem global_commit para todos os participantes, esta troca é representada no estado PhaseTwo, portanto o nome do estado após a etiqueta vote_commit é TwoPhase.

A outra opção de mensagem é representada na linha seguinte, começa com a expressão 1, porque o coordenador espera por apenas uma mensagem vote_abort antes de avançar. Após a expressão é declarada a etiqueta vote_abort seguida do nome do estado por onde a comunicação continua. Quando o coordenador recebe uma mensagem vote_abort decide abortar a transação e envia imediatamente uma mensagem global_abort para todos os participantes, esta troca é representada no estado TwoPhase portanto é este o estado que aparece após a etiqueta vote_abort .

No fim da lista de opções é definida a opção **timeout** tal como no estado anterior. Quando o coordenador não recebe n mensagens vote_commit ou uma mensagem vote_abort ao fim de algum tempo, ocorre um *timeout* e o coordenador decide abortar a transação, portanto envia mensagens global_abort para todos os participantes. Esta troca de mensagens é representada no estado PhaseTwo, portanto no fim da opção **timeout** é indicado este estado.

Por fim, o último estado, ao qual foi dado o nome PhaseTwo, representa o envio da

decisão global do coordenador para os participantes. Este estado é muito semelhante ao estado PhaseOne, sendo que nesta troca o participante c é o emissor e o conjunto ps o recetor. Neste estado podem ser enviadas duas opções de mensagens, global_commit ou global_abort . Quando a decisão é recebida pelos participantes o protocolos termina, por isso após ambas as opções aparece a palavra reservada **end**, indicando que se estas opções forem selecionadas o protocolo termina.

Após estas opções é representada a opção **timeout** um pouco diferente das descritas anteriormente. No protocolo 2PC quando os participantes não recebem a decisão global do coordenador, pedem ajuda a outro participante para saber qual é a decisão. Como o pedido de decisão de um participante a outro não envolve o coordenador as trocas de mensagens referentes a este pedido podem ser representadas noutro protocolo exclusivo para este caso. Para especificar estas trocas foi criado o protocolo Decision, apresentado na figura 3.7. Para que o participante que fez *timeout* seja usado nas trocas de mensagens do protocolo Decision é necessário criar um identificador específico para este participante.

Desta forma dentro dos parênteses da opção **timeout** é definido o identificador p, cujo papel é Participant . No fim desta opção é identificado o nome do estado onde o protocolo continua, como este continua num estado de outro protocolo é usado o nome do protocolo e os identificadores que fazem parte da assinatura do protocolo. Na assinatura do protocolo Decision apenas é declarado o participante que fez *timeout*, por isso apenas se tem que passar o identificador p. Após o nome do protocolo é identificado o nome do estado onde continua a comunicação. No protocolo Decision o estado onde o participante envia o pedido de decisão a outro participante tem o nome AskDecision, portanto no fim da opção **timeout** aparece a expressão Decision(p).AskDecision. Na última linha do protocolo são fechadas as chavetas globais dando o protocolo como terminado.

Protocolo global de pedido de decisão. Na figura 3.7 é apresentado o protocolo global Decision, este protocolo representa o caso em que um participante não recebeu a decisão global do coordenador e precisa de pedir a decisão a outro participante para conseguir terminar. O protocolo começa com a definição dos papéis na linha 1, neste caso apenas é declarado o papel Participant , pois todos os participantes envolvidos no protocolo têm este papel. Na linha 3 é definido o cabeçalho tal como no protocolo anterior. Na assinatura do protocolo apenas é declarado o participante que fez *timeout*, identificado por p cujo papel é Participant , a aridade é omitida pois p representa um único participante.

O corpo do protocolo Decision consiste em dois estados, o estado AskDecision e AnswerDecision. O estado AskDecision apresenta uma troca de mensagens um pouco diferente das do protocolo TwoPhaseCommit, pois é necessário escolher um participante do conjunto ps para comunicar com p. Para definir este participante é usada a palavra reservada **exists** seguida do identificador e papel do novo participante, neste caso foi escolhido o identificador q que tem o papel Participant . Com o participante q declarado passa-se à

troca de mensagens em si.

Nesta troca p é o emissor e q o recetor. De seguida é definida a lista de opções de mensagens que podem ser enviadas neste estado. Neste estado apenas existe a opção de mensagem decision_request . Esta opção leva ao estado AnswerDecision, mas este estado é um pouco diferente pois é necessário passar o identificador do participante q para que este possa ser utilizado na troca de mensagens desse estado. Para tal no fim da opção decision_request após o nome do estado AnswerDecision é passado o identificador q entre parênteses.

```
1  role Participant
2
3  global protocol Decision(p:Participant){
4      AskDecision:
5          exists q:Participant. message p → q{
6              decision_request(): AnswerDecision(q)
7          }
8      AnswerDecision(q:Participant):
9          message q → p {
10             dec_global_commit(): end
11             dec_global_abort(): end
12             no_decision(): AskDecision
13             timeout(): AskDecision
14         }
15 }
```

Figura 3.7: Protocolo global do pedido de decisão entre participantes.

Neste estado não existe a opção **timeout**, pois cada participante tem uma *thread* dedicada para receber pedidos de decisão dos outros participantes. Desta forma cada participante está a executar simultaneamente o protocolo 2PC normal ao mesmo tempo que espera por pedidos de decisão de outros participantes. Portanto os participantes podem ficar bloqueados neste estado até receberem um pedido de outro participante.

O estado seguinte, cujo nome é AnswerDecision, representa o envio da decisão do participante q para o participante p. Este estado é declarado de forma um pouco diferente, pois após o nome do estado é identificado o participante q entre parênteses. O participante q é declarado tal como é feito na assinatura do protocolo, usando o identificador e o seu papel, ou seja, q: Participant . Na troca de mensagens deste estado q é o emissor e p o recetor.

No protocolo 2PC o participante q pode enviar três opções de mensagens, dec_global_commit, dec_global_abort ou no_decision. Quando o participante p recebe uma mensagem dec_global_commit ou dec_global_abort o protocolo termina. Assim na lista de opções de mensagens após estas duas opções aparece a palavra reservada **end**, indicando que o protocolo termina caso estas opções sejam selecionadas.

Como referido no protocolo 2PC quando o participante p recebe uma mensagem no_decision escolhe outro participante para pedir a decisão, assim no fim da opção no_decision é indicado o estado AskDecision. No fim da lista de opções é definida a opção **timeout** e tal como acontece na opção no_decision volta-se ao estado AskDecision para escolher outro participante para p pedir a decisão.

3.4 Regras de tradução

Cada protocolo global origina tantos protocolos locais quanto o número de parâmetros da assinatura do protocolo global mais os participantes declarados através da palavra reservada **`exists`**. Para projetar um protocolo global nos protocolos locais são utilizadas regras de tradução, que permitem transformar automaticamente o protocolo global nos respetivos protocolos locais. Nesta regras são chamadas funções de projeção sobre o protocolo global que originam partes do protocolo local, a função `project` projeta o início do protocolo global, a função `projectBody` projeta as operações de mensagens que são representadas pelo símbolo não terminal `M` na sintaxe dos protocolos globais e por fim a função `projectBranch` projeta as opções de mensagens.

```
project: G rid → L
(1) project (global protocol pid sig M, rid_j, sig) =
    = local protocol pid_rid_j sig projectBody(M,rid_j,sig) if 1<=j<=n
```

A regra nº1 consiste na transformação do cabeçalho do protocolo global para o protocolo local. Nesta tranformação as palavras reservadas **`global protocol`** são substituídas por **`local protocol`**, indicando que o protocolo é local e não global. Ao nome do protocolo global é acrescentado o identificador do participante a quem o protocolo local pertence. A assinatura do protocolo representado por `sig` no protocolo, apresenta a forma $\texttt{sig = (rid}_1\texttt{:role}_1\ \texttt{A}_1\texttt{,..,rid}_n\texttt{:role}_n\ \texttt{A}_n\texttt{)}$ em ambos os tipos de protocolos. A tradução do corpo do protocolo representado pelo símbolo não terminal `M` é explicada em detalhe nas regras seguintes. A expressão `if i<=j<=n` no fim da regra indica que o protocolo global é traduzido em `n` protocolos locais, sendo que a variável `n` representa o número de participantes que são declarados no protocolo global e a variável `j` representa cada um desses participantes.

A troca de mensagem, representada pela palavra reservada **`message`** no protocolo global pode ser traduzida em quatro palavras reservadas diferentes referidas na sintaxe. A forma como a palavra reservada **`message`** é traduzida depende da posição do participante a quem o protocolo local pertence, ou seja, se está antes ou depois da seta que indica a direção da mensagem. Para além da posição também depende se o participante está a comunicar com apenas um participante ou um conjunto de participantes. Na figura 3.8 são apresentadas as várias formas de envio/receção de mensagens que as trocas de mensagens do protocolo global podem originar no protocolo local do participante *a*.

```
(2) projectBody(message rid_1 → rid_2 {e:B_1: M,..,e:B_n: M, e:timeout(rid_3
    :role_3 A_3): M}, rid_1, sig) =
    = send rid_2 { projectBranch(B_1: M, rid_1, sig),..,projectBranch(B_n:
       M, rid_1, sig)} if arity(sig(rid_2))=1
```

Esta regra consiste na tradução de uma troca de mensagens no protocolo global que envolve um participante identificado por $\texttt{rid}_1$ e outro participante $\texttt{rid}_2$. A declaração no fim da regra, $\texttt{arity(sig(rid}_2\texttt{))} = 1$, indica que o participante identificado por $\texttt{rid}_2$ é

um único participante. Quanto à aridade do participante $\texttt{rid}_1$ nada é especificado, por isso tanto pode ser um participante ou um conjunto de participantes.

Protocolo Global	Protocolo Local
message a -> b	**send** b
message a -> bs	**multisend** b
message b -> a	**receive** b
message bs -> a	**multireceive** b

Figura 3.8: Tradução das trocas de mensagens do protocolo global para o protocolo local do participante *a*.

Projetando esta troca no protocolo local do participante $\texttt{rid}_1$ é obtida a palavra reservada **`send`**, que como referido anteriormente, consiste no envio de uma mensagem de um participante para outro participante. A palavra reservada **`send`** é seguida do identificador do participante para quem a mensagem é enviada.

Na tradução as expressões, representadas pelo símbolo não terminal `e`, que antecedem as etiquetas das opções de mensagens não são representadas no protocolo local de $\texttt{rid}_1$, pois as expressões não são aplicadas à receção e para além disso o recetor só irá receber uma mensagem pois o participante $\texttt{rid}_1$ tem aridade 1. A opção **`timeout`** também não é representada no protocolo local de $\texttt{rid}_1$ pois apenas diz respeito à receção da mensagem. Por fim, as opções de mensagens são traduzidas usando a regra nº10 (ver mais à frente).

(3) projectBody(**message** $\texttt{rid}_1 \rightarrow \texttt{rid}_2$ {e:B_1: M,..,e:B_n: M, e:**timeout**($\texttt{rid}_3$:$\texttt{role}_3$ A_3): M}, $\texttt{rid}_1$, sig) =
= **multisend** $\texttt{rid}_2$ {projectBranch(B_1: M, $\texttt{rid}_1$, sig),..,projectBranch(B_n: M, $\texttt{rid}_1$, sig)} if arity(sig($\texttt{rid}_2$))>1

Nesta regra é representada a tradução de uma troca de mensagens no protocolo global entre o participante $\texttt{rid}_1$, que pode ter qualquer valor de aridade, e o participante $\texttt{rid}_2$ que segundo a expressão $\texttt{arity(sig(rid}_2\texttt{))} > 1$ tem aridade superior a um, ou seja, $\texttt{rid}_2$ representa um conjunto de participantes.

Projetando essa troca no protocolo local do participante $\texttt{rid}_1$ obtém-se um **`multisend`** , pois o participante envia múltiplas mensagens, uma para cada participante pertencente ao conjunto recetor. A seguir à palavra reservada **`multisend`** aparece o identificador do conjunto recetor, indicando que o envio da mensagem é feito para esse conjunto. Após o identificador aparece a lista de mensagens que podem ser enviadas neste estado, que são traduzidas utilizando a regra nº 10. Tal como na regra anterior o **`timeout`** e as expressões que precedem as etiquetas das opções de mensagens não aparecem no protocolo local de

rid_2, pois apenas são importantes para o lado recetor das mensagens.

```
(4) projectBody(message rid1 → rid2 {B1: M,..,Bn: M, e:timeout(rid3:
    role3 A3): M}, rid2, sig) =
    = receive rid1 {projectBranch(B1: M, rid2, sig),..,projectBranch(Bn
        : M, rid2, sig), projectBranch(timeout(rid3:role3): M, rid2, sig
        )} if arity(sig(rid1))=1
```

Na regra nº4 representamos uma troca de mensagens de o participante rid_1, que tem aridade 1 tal como indicado pela expressão $arity(sig(rid_1)) = 1$ no fim da regra, e o rid_2, que pode ter qualquer valor de aridade. Projetando esta troca no protocolo local de rid_2 é obtido um **receive**, que consiste na receção de uma mensagem enviada por um participante para outro participante. Neste caso não são utilizadas expressões antes da etiqueta das mensagens, pois como o emissor é apenas um, nesta troca rid_2 apenas recebe uma mensagem. As opções de mensagens são traduzidas usando a regra nº 10 e neste caso a opção **timeout** aplica-se e é traduzida usando a regra nº11 (ver mais à frente).

```
(5) projectBody(message rid1 → rid2 {e:B1: M,..,e:Bn: M, e:timeout(rid3
    :role3 A3): M}, rid2, sig) =
    = multireceive rid1 {e:projectBranch(B1: M, rid2, sig),..,e:
        projectBranch(Bn: M, rid2, sig), projectBranch(timeout(rid3:role
        3): M, rid2, sig)} if arity(sig(rid1))>1
```

Na regra nº5 é representada uma troca de mensagens no protocolo global entre o conjunto de participantes rid_1, tal como indicado pela expressão $arity(sig(rid_1)) > 1$, e o participante rid_2, que pode ter qualquer valor de aridade. Projetando a troca de mensagens no protocolo local de rid_2 é obtido um **multireceive**, pois o participante rid_2 recebe múltiplas mensagens do conjunto rid_1. Após a palavra reservada **multireceive** aparece o identificador do emissor. Como neste caso o participante rid_2 recebe múltiplas mensagens pode ser necessário definir quantas mensagens de cada opção necessita receber para avançar para o estado seguinte, portanto as expressões antes da etiqueta das mensagens aparecem no protocolo local de rid_2. As opções de mensagens são traduzidas através da regra nº10.

Neste caso como se trata de uma receção o **timeout** é traduzido e a expressão que o precede é mantida no protocolo local de rid_2. Esta expressão identifica quantos participantes têm que fazer *timeout* para avançar para o próximo estado, caso rid_2 seja um conjunto de participantes. Por fim, para traduzir o **timeout** é usada a regra nº11.

```
(6) projectBody(exists rid1:role1.M, rid2, sig) =
    = exists rid1:role1.projectBody(M, rid2, sig)
    projectBody(exists rid1:role1.M, rid1, sig) =
    = projectBody(M, rid1, sig)
```

A regra nº6 consiste na tradução da expressão que permite declarar um participante fora da assinatura do protocolo, neste caso o participante identificado é o rid_1. Supondo que rid_2 é o participante que irá trocar mensagens com rid_1, fazendo a projeção para o protocolo local de rid_2 a expressão **exists** mantêm-se igual a como é declarada no

protocolo global. No protocolo local de $\texttt{rid}_1$ a expressão **exists** é omitida, sendo apenas traduzida a troca de mensagens `M`, pois não faz sentido declarar o participante $\texttt{rid}_1$ dentro do seu próprio protocolo local.

(7) projectBody(pid(rid$_1$,.,rid$_n$).M, rid, _) =
= pid_rid(rid$_1$,.,rid$_n$).projectBody(M, rid, sig)

Na regra nº7 podemos observar a tradução da expressão que identifica outro protocolo global, no qual se prossegue a comunicação. Na tradução para o protocolo local de qualquer um dos participantes que participem na comunicação nesse estado, a expressão é basicamente igual a como é declarada no protocolo global, mas ao nome do protocolo global é adicionado o identificador do participante a quem o protocolo local pertence. Por exemplo no protocolo local de $\texttt{rid}_1$ ficará $\texttt{pid\_rid}_1(rid_1, ., rid_n)$.

(8) projectBody(M$_1$ + M$_2$, rid, sig) =
= projectBody(M$_1$, rid, sig) + projectBody(M$_2$, rid, sig)

A regra nº8 consiste na tradução da expressão em que existem duas alternativas de estados para onde a comunicação pode seguir. Na projeção para os protocolos locais de qualquer uma das duas partes que comunicam, a expressão mantêm-se igual à do protocolo global.

(9) projectBody(**end**,_,_) = **end**

Esta regra indica que a palavra reservada **end**, símbolo do fim do protocolo, mantém-se igual na tradução do protocolo global para local. Desta forma nos estados em que o protocolo global termina, os protocolos locais originados a partir do protocolo global também irão terminar nesses estados.

(10) projectBranch(l(param$_1$,..,param$_n$): M, rid, sig) =
= l(param$_1$,..,param$_n$): projectBody(M, rid, sig)

As etiquetas e parâmetros das opções de mensagens mantêm-se inalteradas na tradução de protocolo global para os protocolos locais de ambas as partes participantes na comunicação. Quando os parâmetros opcionais não existem no protocolo global também não existem no protocolo local.

(11) projectBranch(**timeout**(rid:role A): M, rid, sig) =
= **timeout**(rid:role): projectBody(M, rid, sig)

Na regra nº11 é representada a tradução da opção **timeout** no protocolo global para o protocolo local. Quando o participante identificado no **timeout** é um conjunto de participantes na tradução apenas é identificado o participante a quem o protocolo pertence. Portanto deixamos de ter um conjunto para ter um único participante, que apresentará um identificador diferente do identificador do conjunto. Quando o participante identificado no **timeout** é um único participante a aridade é omitida no protocolo global e na tradução para o protocolo local do participante a opção **timeout** mantém-se inalterada.

3.5 Protocolos locais

Os protocolos locais especificam a comunicação para um participante específico, incluindo apenas as trocas de mensagens nas quais o participante está envolvido. A partir do protocolo global originam-se tantos protocolos locais quanto o número de parâmetros da assinatura mais o número de participantes criados através da palavra reservada **exists**. Para representar a comunicação de um conjunto de participantes é usado apenas um protocolo local, pois todos os participantes desse conjunto comportam-se da mesma forma, assim cada participante que pertence ao conjunto tem uma instância do protocolo local.

Se um participante toma parte em vários protocolos globais, então terá tantos protocolos locais quanto o número de protocolos globais em que participa. De acordo com a regra nº1 na transformação do cabeçalho do protocolo global para protocolo local troca-se a palavra reservada **global** por **local** na declaração do protocolo, ao nome do protocolo é adicionado o identificador do participante e a assinatura mantêm-se igual. Os nomes dos estados do protocolo também se mantêm, mas no protocolo local do participante apenas aparecem os estados em que ele está envolvido.

Quanto às trocas de mensagens, são onde surge a maior modificação. A troca de mensagens especificada pela palavra reservada **message** é traduzida para uma das quatro palavras reservadas especificas para envio/receção. As opções de mensagens a ser trocadas mantêm-se, mas o **timeout** e as expressões que o precedem e às opções de mensagens apenas aparecem na parte da receção de uma troca. Quanto ao nome do estado onde continua a comunicação, geralmente é o mesmo que no protocolo global, exceto no caso em que esse estado não existe. Nesse caso é usado o nome do próximo estado em que o participante está envolvido, seguindo a ordem dos estados do protocolo global.

Para ilustrar a tradução de protocolo global para local recorremos novamente ao protocolo 2PC. A comunicação do protocolo 2PC é especificada em dois protocolos globais, nomeadamente o protocolo global TwoPhaseCommit e o protocolo global Decision.

Protocolos locais da execução normal. No protocolo global TwoPhaseCommit (figura 3.6) são declarados um participante c e um conjunto de participantes ps na assinatura e nenhum participante é declarado fora desta, portanto este protocolo origina apenas dois protocolos locais. Um dos protocolos locais representa a comunicação do participante c e o outro representa a comunicação do conjunto de participantes ps.

Protocolo local do coordenador. O protocolo local do participante c, apresentado na figura 3.9, começa com a declaração dos tipos de papéis. No protocolo local são declarados os papéis Coordinator e Participant tal como no protocolo global, para tal é usada a palavra reservada **role** seguida dos nomes dos papéis, como apresentado na linha 1.

O cabeçalho do protocolo local é traduzido usando a regra nº1, ficando bastante semelhante ao do protocolo global, é apenas substituída a palavra reservada **global** por **local**.

Para além disso, o nome do protocolo local é originado pelo nome do protocolo global ao qual é acrescentado o identificador do participante, neste caso fica TwoPhaseCommit_c. Quanto à assinatura é uma cópia da assinatura do protocolo global.

```
1  role Coordinator, Participant
2
3  local protocol TwoPhaseCommit_c(c: Coordinator, ps: Participant [n|n>=1]){
4      PhaseOne:
5          multisend ps {
6              vote_request(): WaitReady
7          }
8
9      WaitReady:
10         multireceive ps {
11            n: vote_commit(): PhaseTwo
12            1: vote_abort(): PhaseTwo
13            timeout(): PhaseTwo
14         }
15
16     PhaseTwo:
17         multisend ps {
18             global_commit(): end
19             global_abort(): end
20         }
21 }
```

Figura 3.9: Protocolo local da execução normal do coordenador.

Como o coordenador participa em todos os estados do protocolo global, todos estes estados irão aparecer no corpo do seu protocolo local. Caso este não participa-se em todos os estados, os estados em que ele não estaria envolvido seriam omitidos do seu protocolo local.

Na projeção os estados do protocolo local ficam com o mesmo nome que no protocolo global. O primeiro estado, cujo nome é PhaseOne (linhas 4-7), segue a regra de tradução nº3 para transformar a expressão **message** c $\rightarrow$ ps em **multisend** ps. O participante c encontra-se antes da seta na troca de mensagens do protocolo global, o que indica que c está a enviar a mensagem para o participante que aparece após a seta. Como neste caso c envia mensagens para um conjunto de participantes, não é enviada apenas uma mensagem, mas sim n mensagens como indica a aridade de ps definida na assinatura do protocolo. Assim é obtido um **multisend**.

Após a palavra reservada **multisend** aparece o identificador do conjunto recetor, que neste caso é ps. Neste estado apenas existe uma opção de mensagem a ser enviada, usando a regra nº10 para a traduzir é obtida a etiqueta vote_request, seguida do nome do estado WaitReady onde a comunicação irá continuar. Observando esta opção que se encontra na linha 6 podemos verificar que é igual à opção apresentada no protocolo global na linha 6. Como referido na regra nº3 a opção **timeout** não aparece no protocolo local de emissor, pois apenas é relevante para quem recebe a mensagem, por isso é omitida no protocolo local do coordenador.

No segundo estado, denominado WaitReady (linhas 9-14), é usada a regra nº5 para traduzir a troca de mensagens **message** ps $\rightarrow$ c para a forma **multireceive** ps. Nesta troca é obtido um **multireceive** ps, pois o participante c aparece após a seta de direção de mensagem,

indicando que este está a receber mensagens e como está a interagir com um conjunto de participante sabemos que recebe múltiplas mensagens, podendo receber no mínimo n mensagens.

Como se trata de uma receção o **timeout** e as expressões antes das opções de mensagens são mantidas no protocolo local. Assim tal como no protocolo global neste estado o coordenador espera por n mensagens com a etiqueta vote_commit, ou por apenas uma mensagem com a etiqueta vote_abort para avançar para o estado PhaseTwo. Caso nenhuma das opções aconteça ao fim de um determinado periodo de tempo, ocorre um *timeout* e o coordenador avança para o estado PhaseTwo. Usando a regra nº5 que indica que as expressões que precedem as opções de mensagens são mantidas, a regra nº10 que traduz as opções de mensagens e a regra nº11 que traduz a opção **timeout** é obtida a lista de opções de mensagens do protocolo local, que é um cópia da lista apresentada no protocolo global.

No último estado, cujo nome é PhaseTwo (linhas 16-20), obtém-se um **multisend** ps tal como no estado PhaseOne. Para traduzir a lista de opções de mensagens é usanda a regra nº10, sendo obtidas as mesmas opções de mensagens que no protocolo global. Ambas as opções levam à terminação do protocolo como indica a palavra reservada **end**, que fica igual na tradução global-local, tal como é indicado pela regra nº9.

Quanto à opção **timeout**, como referido na regra nº3 o **timeout** apenas diz respeito ao lado da receção da troca de mensagens, por isso como neste troca o participante c é o emissor esta opção não é traduzida para o seu protocolo local.Por fim fecham-se as chavetas globais indicando a terminação do protocolo local TwoPhaseCommit_c.

Protocolo local dos participantes. Com o lado da comunicação referente ao coordenador especificado passamos para o lado da comunicação referente aos participantes. Na figura 3.10 está representado o protocolo local do conjunto de participantes, denominado TwoPhaseCommit_ps. Inicia-se com a declaração dos papéis que participam no protocolo na linha 1, tal como no protocolo global são definidos dois papéis, Coordinator e Participant .

O cabeçalho é traduzido usando a regra nº1 tal como no protocolo local anterior, ao nome do protocolo global é adicionado o identificador ps, identificando que este protocolo local pertence ao conjunto de participantes ps. A assinatura do protocolo mantém-se inalterada ao contrário do corpo do protocolo, que sofre algumas mudanças na tradução de protocolo global para local.

O protocolo local TwoPhaseCommit_ps apresenta os mesmos estados que o protocolo global TwoPhaseCommit, porque ps está envolvido em todos os estados do protocolo global.

No estado PhaseOne (linhas 4-8) é usada a regra de tradução nº4 para projectar a troca de mensagens **message** c $\rightarrow$ ps no protocolo local de ps, sendo obtido um **receive** c. Como este protocolo representa a comunicação individual de cada um dos participantes do conjunto a troca de mensagens é de um para um, ou seja, entre o coordenador e um participante. Portanto é obtido um **receive** pois trata-se da receção de uma única mensagem.

Seguindo a regra nº10 a única opção de mensagem deste estado, cuja etiqueta é vote_request , fica igual à do protocolo global. Como o conjunto ps está a receber as mensagens tem de existir a opção de fazer **timeout** para não ficar bloqueado, assim o **timeout** existente no protocolo global é traduzido para o protocolo local de ps. A opção **timeout** é projetada através da regra nº11, originando a palavra reservada *timeout* seguida do estado WaitReady.

```
1  role Coordinator, Participant
2
3  local protocol TwoPhaseCommit_ps(c: Coordinator, ps: Participant [n|n>=1]){
4      PhaseOne:
5          receive c {
6              vote_request(): WaitReady
7              timeout(): WaitReady
8          }
9
10     WaitReady:
11         send c {
12             vote_commit(): PhaseTwo
13             vote_abort(): PhaseTwo
14         }
15
16     PhaseTwo:
17         receive c {
18             global_commit():end
19             global_abort(): end
20             timeout(p:Participant): Decision_p(p).AskDecision
21         }
22 }
```

Figura 3.10: Protocolo local da execução normal dos participantes.

No estado seguinte, denominado WaitReady (linhas 10-14), é usada a regra nº2 para traduzir a troca de mensagens **message** ps → c do protocolo global, obtendo-se um **send** c, pois cada participante do conjunto ps envia uma mensagem para c. Como definido na regra nº2, como se trata do protocolo local do emissor a opção *timeout* e as expressões que precedem as opções de mensagens não são traduzidas para o protocolo local de ps. Desta forma apenas são projetadas as opções de mensagens, usando a regra nº10, ficando iguais às opções de mensagens apresentadas no estado do protocolo global.

No último estado, cujo nome é PhaseTwo (linhas 16-21), é usada a regra nº4 para traduzir a troca de mensagens **message** c → p para **receive** c, tal como no estado PhaseOne. A lista de opções de mensagens deste estado é uma cópia da lista do estado PhaseTwo do protocolo global. Ambas as opções são seguidas pela palavra reservada **end**, que fica igual na tradução de global para local tal como é indicado pela regra nº9.

Como indica na regra nº4 a opção *timeout* do protocolo global é traduzida para o protocolo local do recetor. Para tal é seguida a regra nº11, obtendo-se a palavra reservada **timeout** seguida de parênteses, dentro dos quais é declarado um identificador individual para o participante a quem o protocolo local pertence, pois neste caso não queremos referir ao conjunto ps, mas sim a um participante desse conjunto.

No fim desta opção aparece o estado em que a comunicação continua, neste caso esta continua num estado de um protocolo diferente. Portanto é usada a regra nº7 para

traduzir a invocação do estado AskDecision do protocolo Decision para o protocolo local. Na tradução desta invocação ao nome do protocolo é acrescentado o identificador do participante a quem o protocolo local pertence, assim neste caso fica Decision_p. Após o nome do protocolo é passado o identificador p, que é o participante identificado na opção **timeout**. De seguida é identificado o nome do estado onde a comunicação continua, que neste caso é o estado AskDecision.

Protocolos locais do pedido de decisão. O protocolo global Decision (figura 3.7) tem um participante declarado na assinatura e outro declarado através da palavra reservada **exists**, portanto é projetado em dois protocolos locais. Um dos protocolos locais representa a comunicação do participante p e o outro protocolo local representa a comunicação do participante q. Cada protocolo local representa um lado das trocas de mensagens especificadas no protocolo global, juntando os dois protocolos locais obtém-se a comunicação global representada no protocolo global Decision.

Protocolo local do pedido de decisão. O protocolo local do participante p é apresentado na figura 3.11. Tal como os outros protocolos locais começa com a declaração inicial dos papéis. Esta declaração fica como no protocolo global, portanto após a palava reservada *role* é declarado o papel Participant .

```
1  role Participant
2
3  local protocol Decision_p(p:Participant){
4      AskDecision:
5          exists q:Participant. send q {
6              decision_request(): AnswerDecision(q)
7          }
8
9      AnswerDecision(q:Participant):
10         receive q {
11             dec_global_commit(): end
12             dec_global_abort(): end
13             no_decision(): AskDecision
14             timeout(): AskDecision
15         }
16 }
```

Figura 3.11: Protocolo local de pedido de decisão.

O cabeçalho do protocolo começa como nos protocolos locais anteriores e ao nome do protocolo global é acrescentado o identificador p, portanto o nome fica Decision_p. A assinatura fica igual à do protocolo global, na qual é apenas declarado o participante p.

Após a assinatura é declarado o corpo do protocolo local, que apresenta dois estados, pois o participante p participa em todas as trocas de mensagens do protocolo global. No primeiro estado, denominado AskDecision (linhas 4-7), a expressão **exists** q: Participant é traduzida recorrendo à regra nº6, que define que o formato da expressão é o mesmo que no protocolo global, portanto esta fica igual. Assim através desta expressão é criado o participante q que irá comunicar com o participante p.

De seguida é definida a troca de mensagens deste estado, em que a expressão **message** p → q é transformada num **send** q, usando a regra nº2. Neste estado apenas existe a opção de mensagem decision_request , para a traduzir é usada a regra nº10, que indica que a opção fica igual à do protocolo global. Esta opção leva ao estado AnswerDecision, após o nome deste estado é passado o identificador q, para que este participante possa ser identificado na troca de mensagens desse estado.

O segundo estado, denominado AnswerDecision (linhas 9-15) começa com o nome do estado seguido da declaração do participante q entre parênteses, usando o identificador q e o papel Participant , tal como no protocolo global. Depois recorrendo à regra nº4, a troca de mensagens **message** q → p é tranformada em **receive** q.

De seguida é definida a lista de opções de mensagens, cada opção da lista é projetada usando a regra nº10, sendo que as opções obtidas ficam com o mesmo formato que as opções do protocolo global. As opções dec_global_commit e dec_global_abort são seguidas pela palavra reservada **end**, que fica igual na tradução global-local tal como é indicado pela regra nº9. A opção no_decision é seguida pelo nome do estado AskDecision, indicando que a comunicação volta a esse estado onde vai ser selecionado outro participante a quem p irá pedir a decisão.

Como neste estado o participante p está a receber uma mensagem, seguindo a regra nº4 a opção **timeout** é projectada no protocolo local de p. Para tal recorre-se à regra nº11 que indica que neste caso a opção **timeout** fica igual à apresentada no protocolo global. Portanto a opção **timeout** do protocolo local consiste na palavra reservada **timeout**, seguida do nome do estado AskDecision, indicando que a comunicação volta a esse estado, tal como na opção no_decision.

Protocolo local de resposta a pedidos de decisão. Para representar a parte da comunicação do protocolo global Decision referente ao participante q é apresentado o protocolo da figura 3.12. Este protocolo local representa a comunicação do ponto de vista do participante q, que é o participante que recebe um pedido de decisão do participante p, que não consegue decidir se completa ou aborta a transação.

```
1  role Participant
2
3  local protocol Decision_q(p:Participant){
4      AskDecision:
5          receive p {
6              decision_request(): AnswerDecision
7          }
8
9      AnswerDecision:
10         send p{
11             dec_global_commit(): end
12             dec_global_abort(): end
13             no_decision(): AskDecision
14         }
15 }
```

Figura 3.12: Protocolo local de resposta a pedidos de decisão.

Como em todos os protocolos locais a declaração inicial dos papéis fica igual à declaração apresentada no protocolo global, neste caso apenas é declarado o papel textitParticipant. O cabeçalho do protocolo local também é semelhante ao dos protocolos anteriores, sendo que este tem o nome Decision_q.

O participante q participa em todos os estado da comunicação do protocolo global Decision, então o seu protocolo local apresenta os mesmo estados que o protocolo global. No primeiro estado, denominado AskDecision (linhas 4-7), o participante q recebe uma mensagem de pedido de decisão do participante p.

Segundo a regra nº6 o **exists** que aparece no protocolo global no início da troca de mensagens não faz parte do protocolo local de q, pois não faz sentido um participante declarar-se dentro do seu próprio protocolo. Quanto à expressão **message** p $\rightarrow$ q, como q envia uma mensagem para p, seguindo a regra nº4 a expressão é transformada em **receive** p.

De seguida é definida a lista de opções de mensagens. Neste estado apenas existe a opção decision_request , que direciona a comunicação para o estado AnswerDecision, mas neste caso após o nome do estado não é passado o identificador q como acontece no protocolo global. Isto acontece porque no estado AnswerDecision não é necessário usar o identificador q na troca de mensagens, portanto não é necessário passá-lo como argumento do estado.

No segundo estado, AnswerDecision (linhas 9-14), o nome do estado sofre uma alteração em relação ao apresentado no protocolo global, pois como este é o protocolo local de q não é necessário passar o q como argumento do nome do estado. Através da regra nº2, a troca de mensagens **message** q $\rightarrow$ p é transformada em **send** p.

De seguida é definida a lista de opções de mensagens, segundo a regra nº10 as opções de mensagens ficam iguais na tradução global-local. Assim neste estado existem três opções de mensagens que o participante q pode enviar. A primeira e segunda opção são seguidas pela palavra reservada **end**, que como indica a regra nº9 fica igual na projeção global-local. A terceira opção direciona a comunicação para o estado AskDecision, indicando que quando q também não consegue decidir volta para o estado AskDecision, onde espera por mais pedidos de decisão.

Como referido na regra nº4 a opção **timeout** apenas é importante para a receção, portanto não é traduzida para o protocolo local de q.

Toda a comunicação do coordenador é especificada no protocolo local TwoPhaseCo mmit_c, enquanto que a comunicação dos participantes é especificada em três protocolos diferentes, o TwoPhaseCommit_ps, Decision_p e Decision_q. Cada participante tem uma instância de cada um destes protocolos que pode ser invocada durante a comunicação.

3.6 Considerações finais

Neste capítulo apresentámos a linguagem para construir os protocolos globais e locais. Para ilustrar a construção de ambos os tipos de protocolos recorremos ao protocolo 2PC, que é um protocolo bastante conhecido e fácil de perceber. Foram apresentadas as regras de tradução que permitem projetar os protocolos globais em protocolos locais.

Como podem observar basta aplicar as regras de tradução, apresentadas na secção 3.4, para projetar um protocolo global em vários protocolos locais. Os protocolos locais apresentam um formato muito semelhante ao dos protocolos globais, o que facilita bastante a tradução global-local.

Nos protocolos exemplo apresentados não foram utilizadas todas as formas apresentadas na sintaxe, mas as que não foram utilizadas são explicadas em detalhe nos protocolos dos casos de estudo apresentados no capítulo seguinte.

Capítulo 4

Casos de estudo

Para mostrar que a nossa linguagem é poderosa e consegue representar vários sistemas distribuídos tolerantes a faltas apresentamos dois casos de estudo. O primeiro chama-se *Three-Phase Commit*, que é uma variante do protocolo *Two-Phase Commit* e o segundo caso de estudo chama-se *Viewstamped Replication*. Para cada caso de estudo construímos protocolos globais e projetámo-los para protocolos locais usando as regras de tradução.

4.1 Protocolo Three-Phase Commit

O protocolo *Two-Phase Commmit* (2PC) tem o problema que quando o coordenador falha os participantes podem não conseguir decidir, ficando bloqueados até que o coordenador recupere. Em 1981, Skeen desenvolveu o protocolo *Three-Phase Commit* (3PC) [12], que evita que os participantes fiquem bloqueados quando ocorrem falhas. O 3PC é bastante semelhante ao 2PC, apenas contém mais uma fase de preparação do *commit* da transação antes da fase de decisão final.

Tal como no 2PC existe um processo coordenador e vários processos participantes. Na figura 4.1 são representadas as mensagens trocadas neste protocolo entre o coordenador e os participantes, excluindo a comunicação feita entre participantes no caso em que o coordenador falha. No 3PC o coordenador começa por enviar uma mensagem vote_request para todos os participantes e depois fica à espera das respostas. Se algum participante votar para abortar a transação, a decisão final será abortar, então o coordenador envia uma mensagem global_abort para os participantes. No entanto, quando todos os participantes votam para fazer *commit* da transação o coordenador envia uma mensagem prepare_commit para os participantes. Se os participantes estiverem prontos para fazer *commit* da transação enviam uma mensagem ready_commit para o coordenador. Só quando todos os participantes estiverem prontos é que o coordenador envia a mensagem global_commit.

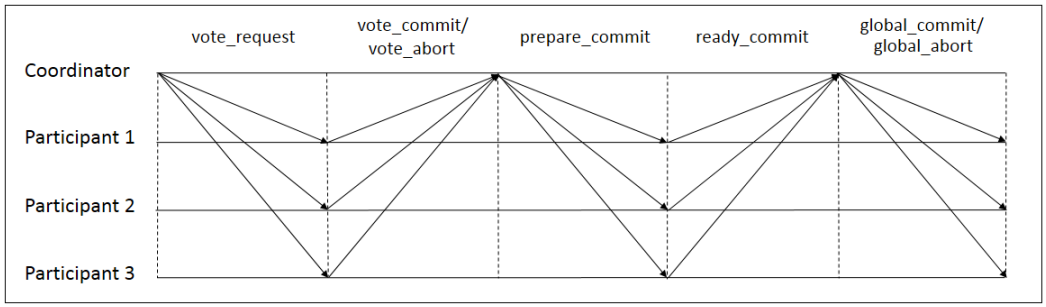

Figura 4.1: Trocas de mensagens realizadas no protocolo Three-Phase Commit quando não ocorrem falhas.

No 3PC também existem algumas situações em que um processo fica bloqueado enquanto espera pela chegada de mensagens. A primeira situação acontece quando o participante fica à espera da mensagem vote_request vinda do coordenador. Neste caso o participante assume que o coordenador falhou, então decide abortar a transação e envia uma mensagem vote_abort para o coordenador, tal como no protocolo 2PC. Outra situação ocorre quando o coordenador espera pelos votos dos participantes, se ocorrer um *timeout* ele assume que um participante falhou e aborta a transação enviando uma mensagem global_abort para todos os participantes.

Numa situação em que o coordenador está bloqueado à espera das mensagens ready_commit, se ocorrer um *timeout* o coordenador assume que um participante falhou. No entanto sabe-se que esse participante votou para fazer *commit*, então o coordenador pode enviar a decisão final global_commit para os participante que estão a funcionar. Os participantes que falham dependem de um protocolo de recuperação para fazerem *commit* da sua parte da transação quando voltam a ficar funcionais.

Outro situação ocorre quando um participante P fica bloqueado à espera das mensagens prepare_commit ou da decisão final vindas do coordenador. Se ocorrer um *timeout*, P assume que o coordenador falhou e contacta outro participante (Q), para descobrir qual é a decisão. Se o participante Q recebeu a decisão final do coordenador envia-a para P, que faz *commit* ou aborta a transação de acordo com a decisão recebida. No caso em que Q está à espera da decisão final é seguro fazer *commit* da transação, pois para Q estar neste estado é porque já todos os participantes decidiram fazer *commit* anteriormente.

Caso Q ainda se encontre à espera da mensagem vote_request , é seguro abortar a transação, portanto Q envia uma mensagem global_abort para P. Isto acontece pois se Q ainda se encontra à espera do pedido de voto nenhum dos outros participantes pode ter recebido a mensagem prepare_commit que indica que todos votaram para fazer *commit* da transação, porque para isso acontecer Q tinha que ter votado. Assim se nenhum participante recebeu o resultado da votação podem abortar a transação sem problemas.

Por fim, se Q já votou para fazer *commit* e está à espera da mensagem prepare_commit , ele não sabe qual é decisão final, portanto envia uma mensagem no_decision para P.

Neste caso *P* contacta outro participante e caso todos os participantes estejam à espera da mensagem prepare_commit a transação deve ser abortada.

4.1.1 Protocolos globais

Para representar a comunicação global do protocolo 3PC foram criados dois protocolos, tal como no 2PC. O primeiro protocolo é denominado ThreePhaseCommit e representa a execução normal do protocolo em que o coordenador e os participantes trocam mensagens. O segundo protocolo tem o nome Decision e representa o pedido de decisão entre participantes.

Protocolo global da execução normal. Começando pelo protocolo ThreePhaseCommit, apresentado na figura 4.2, este protocolo é bastante semelhante ao TwoPhaseCommit (figura 3.6), apenas apresenta mais dois estados que consistem na preparação do *commit*. O protocolo começa com a declaração inicial dos papéis na linha 1, neste caso existem dois papéis, Coordinator e Participant .

```
1  role Coordinator, Participant
2
3  global protocol ThreePhaseCommit(c: Coordinator, ps: Participant [n|n>=1]){
4      PhaseOne:
5          message c → ps {
6              vote_request(): WaitReady
7              timeout(): WaitReady
8          }
9
10     WaitReady:
11         message ps → c{
12             n: vote_commit(): PhaseTwo
13             1: vote_abort(): PhaseThree
14             timeout(): PhaseThree
15         }
16
17     PhaseTwo:
18         message c -> ps{
19             prepare_commit(): Prepare
20             timeout(p:Participant): Decision(p).AskDecision
21         }
22
23     Prepare:
24         message ps -> c {
25             n: ready_commit(): PhaseThree
26             timeout(): PhaseThree
27         }
28
29     PhaseThree:
30         message c → ps{
31             global_commit(): end
32             global_abort(): end
33             timeout(p:Participant): Decision(p).AskDecision
34         }
35 }
```

Figura 4.2: Protocolo global da execução normal do protocolo 3PC.

De seguida é declarado o cabeçalho do protocolo que é igual ao do TwoPhaseCommit , apenas se altera o nome do protocolo para ThreePhaseCommit. Assim na assinatura do

protocolo é declarado o participante c cujo papel é Coordinator e o conjunto de participantes ps, que têm o papel Participant .

O corpo deste protocolo apresenta cinco estados, que representam as trocas de mensagens entre o participante c e o conjunto de participantes ps necessárias para chegar a uma decisão final. Os estados PhaseOne (linhas 4-8) e WaitReady (linhas 10-15) são iguais aos do protocolo TwoPhaseCommit, com a exceção de que no estado WaitReady a opção vote_abort e o **timeout** avançam para o estado PhaseThree. Nesse estado c irá enviar as mensagens de decisão final para abortar a transação.

O estado PhaseTwo (linhas 17-21) consiste no envio das mensagens prepare_commit do participante c para o conjunto de participante ps. A opção de mensagem prepare_commit direciona a comunicação para o estado Prepare. A opção **timeout** deste estado leva a que comunicação avance para o protocolo global Decision, no qual o participante pode pedir a decisão a outro participante. Portanto tal como no protocolo TwoPhaseCommit é declarado o participante p no **timeout**, e no fim desta opção aparece a expressão Decision(p).AskDecision, que indica que a comunicação continua no estado AskDecision do protocolo Decision.

O próximo estado, denominado Prepare (linhas 23-27), consiste no envio das mensagens ready_commit de cada participante do conjunto ps para o participante c. Neste estado c precisa de receber esta mensagem de todos os participantes, assim usar a expressão n antes da etiqueta da mensagem. Tanto esta opção como o **timeout** levam ao estado PhaseThree.

Por fim existe o estado PhaseThree (linhas 29-34) que é igual ao estado PhaseTwo do protocolo global TwoPhaseCommit.

Protocolo global do pedido de decisão. O protocolo global Decision do protocolo 3PC é igual ao do protocolo 2PC que se encontra na figura 3.7.

Como é possível verificar pelos protocolos apresentados acima, o protocolo 3PC é muito semelhante ao protocolo 2PC. Bastou apenas acrescentar dois estados ao protocolo TwoPhaseCommit para obter o protocolo ThreePhaseCommit e quanto ao protocolo Decision não foi preciso fazer qualquer mudança. Com este passo adicional resolve-se o problema do 2PC, desta forma os participantes conseguem sempre chegar a uma decisão mesmo que o coordenador tenha falhado.

4.1.2 Protocolos locais

Nesta secção os protocolos globais do caso de estudo 3PC são projetados em vários protocolos locais que representam a comunicação do ponto de vista de cada um dos participantes da comunicação global. Para fazer esta projeção são utilizadas diversas regras de tradução, que foram definidas anteriormente na secção 3.4.

No protocolo global ThreePhaseCommit (figura 4.2) existe um participante c e um conjunto de participantes ps, portanto a partir deste protocolo global são obtidos dois protocolos locais. Um dos protocolos representa a comunicação do ponto de vista do participante

c e o outro representa a comunicação do ponto de vista de cada participante do conjunto ps.

Protocolo local do coordenador. Na figura 4.3 é apresentado o protocolo local que representa a comunicação do participante c, este protocolo apresenta cinco estados como no protocolo global ThreePhaseCommit, pois o participante c participa em todas as trocas do protocolo global. Este protocolo local é semelhante ao protocolo local TwoPhaseCommit_c apresentado na figura 3.9.

```
1  role Coordinator, Participant
2
3  local protocol ThreePhaseCommit_c(c: Coordinator, ps: Participant [n|n>=1]){
4      PhaseOne:
5          multisend ps {
6              vote_request(): WaitReady
7          }
8
9      WaitReady:
10         multireceive ps {
11            n: vote_commit(): PhaseTwo
12            1: vote_abort(): PhaseThree
13            timeout(): PhaseThree
14         }
15
16     PhaseTwo:
17         multisend ps {
18             prepare_commit(): Prepare
19         }
20
21     Prepare:
22          multireceive ps {
23            n: ready_commit(): PhaseThree
24            timeout(): PhaseThree
25         }
26
27     PhaseThree:
28          multisend ps {
29              global_commit():end
30              global_abort(): end
31         }
32 }
```

Figura 4.3: Protocolo local da execução normal do coordenador.

Seguindo a regra nº1 o cabeçalho do protocolo local fica muito semelhante ao do protocolo global, apenas é substituída a palavra reservada **global** por **local** e ao nome do protocolo global é acrescentado o identificador c, formando assim o nome do protocolo local. A assinatura fica igual à do protocolo global, na qual é declarado o participante c e o conjunto de participantes ps.

O protocolo local do coordenador apresenta os mesmos estados que o protocolo global ThreePhaseCommit, pois este está envolvido em todos os estados do protocolo global. Seguindo as regras de tradução global-local as trocas de mensagens são projetadas em trocas individuais. Como no protocolo global o coordenador comunica sempre com um conjunto de participantes as trocas de mensagens do protocolo local deste participante são sempre do tipo **multisend** ou **multireceive**, dependendo se c se encontra antes ou depois da seta de direção de mensagens no protocolo global. As expressões que precedem as eti-

quetas das mensagens e a opção **timeout** apenas aparecem na lista de opções de mensagens do **multireceive**.

Protocolo local dos participantes. No protocolo local apresentado na figura 4.4 é representada a comunicação do ponto de vista do conjunto de participantes ps. Este protocolo é muito semelhante ao protocolo local TwoPhaseCommit_c apresentado na figura 3.10.

O corpo do protocolo consiste em cinco estados, pois ps está envolvido em todos os estados do protocolo global. Como este protocolo local representa as trocas de mensagens de cada participante os envios ou receções são sempre entre um participante e o coordenador. Assim as trocas de mensagens do protocolo local são do tipo **send** ou **receive**, dependendo se ps se encontra antes ou depois da seta de direção de mensagens no protocolo global.

Neste protocolo não aparecem as expressões que precedem as etiquetas das mensagens, pois essas só se aplicam à operação **multireceive**. A opção **timeout** apenas aparece na lista de opções das operações **receive** e fica sempre igual à do protocolo global.

```
1  role Coordinator, Participant
2
3  local protocol ThreePhaseCommit_ps(c: Coordinator, ps: Participant [n|n>=1]){
4      PhaseOne:
5          receive c {
6              vote_request(): WaitReady
7              timeout(): WaitReady
8          }
9
10     WaitReady:
11         send c {
12             vote_commit(): PhaseTwo
13             vote_abort(): PhaseThree
14         }
15
16     PhaseTwo:
17         receive c {
18             prepare_commit(): Prepare
19             timeout(p:Participant): Decision_p(p).AskDecision
20         }
21
22     Prepare:
23          send c {
24             ready_commit(): PhaseThree
25         }
26
27     PhaseThree:
28         receive c {
29             global_commit():end
30             global_abort(): end
31             timeout(p:Participant): Decision_p(p).AskDecision
32         }
33 }
```

Figura 4.4: Protocolo local da execução normal dos participantes.

Tal como acontece no protocolo global Decision também os protocolos locais do caso de estudo 3PC são iguais aos dos protocolo 2PC, apresentados nas figuras 3.11 e 3.12.

Nesta secção foi apresentado mais um exemplo de como a nossa linguagem é utilizada para contruir protocolos globais e como as regras são aplicadas para obter os protocolos

locais. No entanto, achámos necessário fornecer mais um exemplo num cenário completamente diferente e que explora toda a nossa sintaxe, pois o 3PC é bastante semelhante ao 2PC. Assim na secção seguinte introduzimos outro protocolo mais complexo que irá ajudar a compreender o funcionamento do nosso projeto.

4.2 Protocolo Viewstamped Replication

O protocolo *Viewstamped Replication* (VR) [10] implementa uma técnica de replicação que consegue tolerar falhas por terminação inesperada de um programa em alguma parte do sistema. A replicação de máquinas de estados é definida geralmente através de um conjunto de clientes que submetem comandos a um conjunto de réplicas comportando-se como um serviço centralizado. A implementação deste paradigma requer que três propriedades sejam verdadeiras:

- Estado inicial: todas as réplicas corretas começam no mesmo estado;
- Determinismo: todas as réplicas corretas que recebem o mesmo *input* no mesmo estado geram o mesmo *output* e estado resultante;
- Coordenador: todas as réplicas corretas processam a mesma sequência de comandos.

Estes sistemas apresentam o desafio de assegurar que as operações são executas pela mesma ordem por todas as réplicas quando existem pedidos concorrentes de clientes e falhas.

O protocolo VR assegura fiabilidade e disponibilidade quando podem falhar no máximo f réplicas, para tal necessita de usar um grupo de $2f+1$ réplicas. Neste protocolo existem vários tipos de processos: clientes, primário e réplicas backup. Os clientes enviam pedidos diretamente para o primário, que trata da ordenação dos pedidos. As $2f$ réplicas *backup* aceitam as ordens do primário. Caso o primário falhe uma das réplicas *backup* assume o seu lugar.

O sistema avança por uma série de vistas ao longo do tempo, sendo que em cada vista uma das réplicas é o primário e quando se avança para a próxima vista escolhe-se outra réplica para ser o primário. Cada vista tem um *view-number* associado, que começa a zero e determina qual a réplica que é primário nessa vista. Quando as réplicas detetam que o primário falhou executam um protocolo de troca de primário no qual é escolhido um novo primário.

4.2.1 Descrição do VR

O protocolo VR consiste em três sub-protocolos que em conjunto asseguram que a replicação das operações decorre corretamente. Um dos sub-protocolos representa a operação normal que consiste no processamento dos pedidos dos clientes. Outro sub-protocolo trata das trocas de primário para selecionar o novo primário. Por fim, outro sub-protocolo

trata da recuperação das réplicas que falham para que estas voltem a juntar-se ao grupo, para que o número limite de réplicas falhadas não seja excedido.

Estado dos processos. Cada réplica guarda o seu estado, que consiste nas seguintes variáveis:

- A configuração é um *array* ordenado com os endereços IP (*Internet Protocol*) de cada uma das réplicas.
- A *replicaID* corresponde à posição do *array* onde o IP dessa réplica está guardado.
- O *view-number* atual, que começa a 0.
- O *status* atual, que identifica qual o sub-protocolo que a réplica está a executar. O *status* pode ser *normal*, *view-change* ou *recovering*.
- O *op-number* associado ao último pedido recebido, que começa a 0.
- O *log* é um *array* que tem *op-number* entradas. As entradas contêm os pedidos recebidos por ordem.
- O *commit-number* é o *op-number* da última operação que foi executada.
- A *client-table* guarda o número de pedido mais recente de cada cliente, se esse pedido já tiver sido executado também guarda o resultado do mesmo.

Cada cliente também tem um estado que guarda a configuração e o *view-number* que ele pensa ser o mais recente. Esse *view-number* permite-lhe saber quem é o primário. Para além disso guarda o *clientID* e o *request-number* mais recente. Este *request-number* vai aumentando à medida que o cliente faz pedidos e é usado pelas réplicas para que estas não executem um pedido mais do que uma vez e também serve para o cliente descartar respostas duplicadas aos seus pedidos.

Operação normal. O sub-protocolo que representa a operação normal, descreve como o protocolo VR decorre quando o primário não falha. Neste protocolo o *status* das réplicas tem que ser *normal* para que estas possam processar pedidos dos clientes. Assume-se que todas as réplicas estão na mesma vista e só processam mensagens cujo *view-number* seja igual ao do seu estado.

Na figura 4.5 são representadas as mensagens trocadas entre os diversos processos durante a operação normal, neste exemplo existem três réplicas, assim seguindo a expressão *2f+1* o sistema continua a funcionar se até uma réplica falhar.

No início deste protocolo o cliente envia uma mensagem *request(operation, clientID, requestNumber)* para o primário, onde *request* é a etiqueta da mensagem, que indica que esta mensagem é um pedido do cliente. Esta mensagem tem três parâmetros, o *operation*, que é a operação que o cliente quer executar, o *clientID* e o *request-number* associado ao pedido.

Quando o primário recebe o pedido, compara o *request-number* com o que tem na *client-table*. Se o *request-number* não for maior que o da *client-table* ele ignora-o e envia

uma mensagem de resposta ao cliente que o ajuda a ficar atualizado. Caso seja maior o primário avança o *op-number*, adiciona o pedido ao fim do *log* e atualiza a *client-table* com o novo *request-number* na entrada deste cliente. Depois envia uma mensagem *prepare(viewNumber, messageReceived, opNumber, commitNumber)* para as outras réplicas, informando-as do pedido recebido. Esta mensagem tem a etiqueta *prepare* e quatro parâmetros, o *view-number*, a mensagem de pedido recebida do cliente, o *op-number* atribuído ao pedido e o *commit-number*.

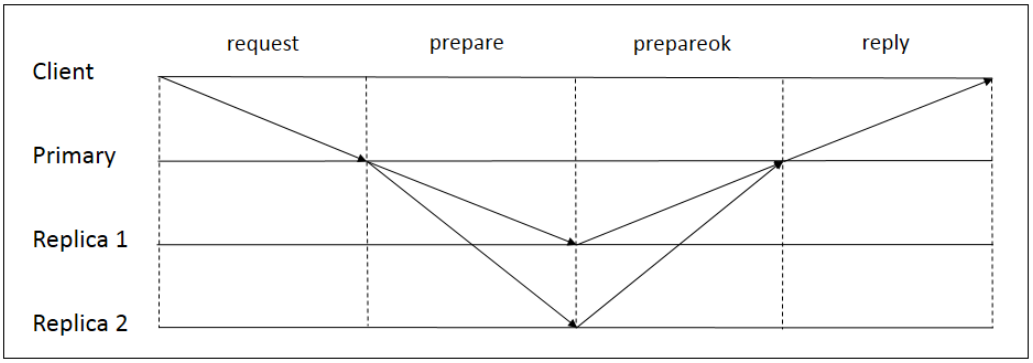

Figura 4.5: Trocas de mensagens realizadas no sub-protocolo que representa a operação normal do protocolo VR.

As réplicas processam as mensagens por ordem, assim não aceitam um *prepare* com um *op-number* até que tenham entradas no *log* para os pedidos anteriores. Quando uma réplica tem todas as entradas dos pedidos anteriores incrementa o *op-number*, adiciona o pedido ao fim do *log* e atualiza a *client-table* com o *request-number* do pedido. Por fim, envia uma mensagem *prepareok(viewNumber, opNumber, replicaID)* para o primário indicando que esta operação e todas as anteriores foram preparadas localmente.

O primário espera por f mensagens *prepareok* de diferentes réplicas, quando recebe todas as mensagens necessárias considera a operação e as anteriores, que têm *op-numbers* inferiores, como *commited*. Depois de ter executado todas as operações anteriores, o primário executa a operação fazendo uma chamada ao código do serviço e incrementa o *commit-number*. De seguida envia uma mensagem *reply(viewNumber, requestNumber, result)* para o cliente. Nesta mensagem são passados como parâmetros o *view-number* atual, o *request-number* que vinha no pedido do cliente e o *result* que é o resultado da chamada. Por fim, o primário atualiza a *client-table* com o resultado da chamada.

As réplicas são informadas sobre o *commit* através do *commit-number* enviado nas mensagens *prepare*. Quando o primário não recebe um novo pedido de um cliente dentro de algum tempo, ele informa as réplicas através da mensagem *commit(viewNumber, commitNumber)*. O primário vai enviando mensagens *commit* para as réplicas em intervalos regulares caso não esteja a receber pedidos dos clientes, para que estas não pensem que este falhou e façam *timeout*.

Quando uma réplica é informada de um *commit* espera até ter o pedido no *log* e até executar todas as operações anteriores. Depois executa a operação fazendo uma chamada ao código do serviço, incrementa o *commit-number*, atualiza a entrada do cliente na *client-table*, mas não envia uma mensagem de resposta ao cliente, pois só o primário é que comunica com o cliente.

Se o cliente não receber a resposta ao fim de algum tempo ele reenvia o pedido para todas as réplicas, para tentar contactar o primário. As réplicas ignoram este pedido, só o primário é que o processa.

Troca de primário. O sub-protocolo de troca de primário é usado para continuar a fornecer serviço quando o primário falha, para tal é escolhida outra réplica para desempenhar o seu papel. Este protocolo é executado quando as réplicas fazem *timeout*, porque não receberam uma mensagem do primário atempadamente.

Neste sub-protocolo quando uma réplica se apercebe da necessidade de mudar de primário, ela avança o *view-number*, muda o seu *status* para *view-change* e envia uma mensagem *startviewchange(viewNumber, replicaID)* para as outras réplicas. Uma réplica percebe que é necessário trocar de primário através do seu temporizador. Para acelerar a troca de primário as réplicas poderiam detetar a necessidade de um troca ao receberem uma mensagem *startviewchange* ou *doviewchange* com um *view-number* superior ao seu *view-number*, mesmo que o seu temporizador não se tenha expirado.

Quando um réplica recebe f mensagens *startviewchange* com o novo *view-number* ela envia uma mensagem *doviewchange(viewNumber, log, latestViewNumber, opNumber, commitNumber, replicaID)* para o novo primário, que é a réplica a quem o novo *view-number* corresponde. Nesta mensagem o parâmetro *latestViewNumber* identifica o *view-number* da última *view* em que o *status* era *normal*.

Quando o novo primário recebe $f+1$ mensagens *doviewchange* de diferentes réplicas, incluindo ele próprio, ele coloca o seu *view-number* igual ao das mensagens e seleciona como novo *log* o da mensagem que tem o maior *latestViewNumber*. Caso várias mensagens tenham o mesmo *latestViewNumber* ele seleciona a mensagem que tem o maior *op-number*. De seguida o primário altera o seu *op-number* para ser igual ao da última entrada do *log*, altera o *commit-number* para o maior que aparece nas mensagens *doviewchange* e altera o seu *status* para *normal*. Por fim, envia uma mensagem *startview(viewNumber, log, opNumber, commitNumber)* para as outras réplicas, na qual é passado o novo *log*. Neste ponto o novo primário começa a aceitar pedidos dos clientes, executa por ordem as operações *committed* que ainda não tinha executado anteriormente e atualiza a *client-table*.

Quando as outras réplicas recebem a mensagem *startview*, substituem o seu log pelo da mensagem, alteram o seu *op-number* para o da última entrada do *log*, alteram o seu *view-number* para o da mensagem, alteram o seu *status* para *normal* e atualizam a

informação da sua *client-table.* Se existem mensagens no *log* que ainda não foram executas, as réplicas enviam uma mensagem *prepareok* para o primário. Depois executam todas as operações que sabem ter sido executadas e que ainda não executaram anteriormente, avançam o seu *commit-number* e atualizam a informação na sua *client-table*.

Uma troca de primário pode não ter sucesso, por exemplo devido à falha do novo primário. Neste caso as replicas começam uma nova troca de primário com outra réplica como primário.

Recuperação após falha. No sub-protocolo de recuperação a réplica que falhou atualiza o seu estado com a ajuda das outras réplicas que se encontram funcionais. Quando uma réplica volta a ficar funcional após falhar altera o seu *status* para *recovering* e durante esta fase a réplica não pode participar no sub-protocolo de processamento de pedidos dos clientes, nem no de troca de primário.

No inicio do sub-protocolo a réplica que está a recuperar envia uma mensagem *recovery(replicaID, nonce)* para todas as outras réplicas, em que o parâmetro *nonce* é um número inteiro aleatório.

Uma réplica apenas pode responder a uma mensagem *recovery* se o seu *status* for *normal*. Nesse caso a réplica envia uma mensagem *recoveryresponse(viewNumber, nonce, log, opNumber, commitNumber, primaryID)* para a réplica que está a recuperar. Quando o primário envia a mensagem *recoveryresponse* ele usa o seu *log*, *op-number* e *commit-number* para preencher os parâmetros da mensagem, no caso de ser uma das réplicas *backup* esses parâmetros são *nil*.

A réplica que está a recuperar espera por receber *f+1* mensagens *recoveryresponse* com o seu *nonce*, incluindo uma mensagem do primário. Depois atualiza o seu estado usando a mensagem que recebeu do primário, altera o seu *status* para *normal* e o protocolo de recuperação está completo.

Transferência de estado. Por vezes as réplicas podem-se atrasar e precisam de fazer uma transferência de estado para se atualizarem. Através das mensagens *prepare* e *commit* a réplica pode perceber que lhe faltam pedidos na *view* atual ou que as outras réplicas já se encontram noutra *view*, verificando assim que precissa de atualizar o seu estado.

Para fazer a transferência de estado a réplica atrasada envia uma mensagem *getstate(viewNumber, opNumber, replicaID)* para outra réplica. Essa réplica responde se o seu *status* for *normal* e está na *view-number* indicada. Nesse caso a réplica responde com uma mensagem *newstate(viewNumber, log, opNumber, commitNumber)*, na qual o log apenas contém as entradas após o *op-number* indicado na mensagem *getstate*, de forma a reduzir o tamanho do *log* que tem que ser enviado. Após receber a mensagem a réplica atrasada atualiza o seu estado usando a mensagem e volta à operação normal. Caso a réplica atrasada não receba a mensagem de resposta atempadamente, ela contacta outra

réplica para a ajudar a ficar atualizada.

A transferência de estado é muito semelhante ao sub-protocolo de recuperação, mas neste caso apenas se contacta uma réplica e na recuperação contactam-se todas as réplicas.

4.2.2 Protocolos globais

Para representar a comunicação do protocolo VR foram criados quatro protocolos globais, um para representar cada um dos sub-protocolos referidos acima e um protocolo para representar a transferência de estado.

Operação normal. O protocolo global VR apresentado na figura 4.6 representa a comunicação do sub-protocolo que consiste na operação normal do sistema, no qual são processados os pedidos dos clientes. Neste protocolo global é representada a interação das réplicas com um cliente. Para cada interação com um cliente diferente existe uma instância deste protocolo.

```
1  type opNumber, commitNumber, requestNumber, viewNumber, latestViewNumber = natural
2  type operation, clientID, replicaID, primaryID, nonce = natural
3  type result, messageReceived, log = ...
4  role Client, Replica
5
6  global protocol VR(c: Client, p: Replica, rs: Replica [2*f|f>0]) {
7      RequestPhase:
8          message c -> p {
9              request(operation, clientID, requestNumber): PreparePhase + ReplyPhase
10             timeout(): CommitPhase
11         }
12
13     PreparePhase:
14         message p -> rs {
15             prepare(viewNumber, messageReceived, opNumber, commitNumber): PrepareOkPhase
16             f: timeout(xs:Replica [x|x>=f]): ViewChange(rs U p, xs).ViewChangePhase
17         }
18
19     PrepareOkPhase:
20         message rs -> p {
21             f: prepareok(viewNumber, opNumber, replicaID): ReplyPhase
22         }
23
24     ReplyPhase:
25         message p -> c {
26             reply(viewNumber, requestNumber, result): end
27             timeout(): RetryPhase
28         }
29
30     RetryPhase:
31         message c -> rs U p {
32             request(operation, clientID, requestNumber): PreparePhase  + ReplyPhase
33         }
34
35     CommitPhase:
36         message p -> rs {
37             commit(viewNumber, commitNumber): end
38             f: timeout(xs:Replica [x|x>=f]): ViewChange(rs U p, xs).ViewChangePhase
39         }
40 }
```

Figura 4.6: Protocolo global da operação normal, que consiste em processar pedidos dos clientes.

No início do protocolo são declaradas várias variáveis que são passadas como parâmetros nas mensagens trocadas no corpo do protocolo. Nos protocolos apresentados anteriormente não existia esta declaração de variáveis, pois as mensagens trocadas não precisavam de parâmetros. Na declaração de variáveis é indicado o seu tipo, quando este depende da escolha do programador indica-se que o tipo é indefinido e é representado por três pontos.

Para declarar as variáveis usa-se a palavra **type** seguida do nome da variável, caso existam várias variáveis do mesmo tipo podem-se declarar todas na mesma linha separando-as por vírgulas. Para indicar o tipo das variáveis é usado o símbolo = seguido de um tipo básico, como por exemplo **natural**, que representa os números inteiros não negativos.

De seguida são declarados os papéis, neste caso existem dois, Client e Replica. O cabeçalho do protocolo é declarado como nos protocolos globais apresentados anteriormente. Na assinatura são declarados os participantes que interagem neste protocolo, o cliente é declarado com o identificador c e papel Client, o primário com o identificador p e papel Replica e por último o conjunto de réplicas *backup* com o identificador rs e papel replica. A aridade do conjunto rs é dada pela expressão 2∗f e a proposição f>0 que indica que f tem que ser maior que zero, portanto este conjunto é constituído por no mínimo duas réplicas.

O protocolo começa com o cliente a enviar um pedido para o primário, quando o primário recebe o pedido existem duas alternativas por onde a comunicação pode continuar. Caso o cliente envie um pedido que já enviou anteriormente o primário envia-lhe uma mensagem reply para o atualizar. Caso o pedido não seja repetido o primário segue a operação normal e envia uma mensagem prepare para cada uma das réplicas.

No estado RequestPhase (linhas 7-11), existe uma troca da mensagem request entre c e p. A opção request pode levar a dois estados distintos, PreparePhase ou ReplyPhase, portanto no fim desta opção aparecem os nomes destes estados separados pelo símbolo +, indicando que a comunicação pode seguir por qualquer uma destas alternativas. Caso ocorra um *timeout* a comunicação avança para o estado CommitPhase.

No estado PreparePhase (linhas 13-17) o primário envia uma mensagem prepare para cada réplica para as informar do pedido recebido. Se a opção prepare for selecionada a comunicação avança para o estado PrepareOkPhase. Neste estado várias réplicas podem fazer *timeout* se a mensagem não chegar atempadamente. Para identificar o conjunto das réplicas que fez *timeout* é definido o identificador xs. Antes da palavra reservada **timeout** é indicada a expressão f que indica que pelo menos f réplicas têm que fazer *timeout* para a comunicação avançar para o estado seguinte.

Na opção **timeout** a comunicação avança para o estado o estado ViewChangePhase do protocolo global ViewChange. O protocolo ViewChange recebe dois identificadores, o primeiro consiste no conjunto rs ao qual se junta p usando o símbolo de união **U**, para representar o conjunto de todas as réplicas, o segundo identificador é xs que representa as réplicas que

fizeram *timeout*.

No estado PrepareOkPhase (linhas 19-22) as réplicas respondem ao primário com uma mensagem prepareok. Nesta troca de mensagens apenas existe a opção prepareok com três parâmetros, que direciona a comunicação para o estado ReplyPhase. Neste estado não existe a opção **timeout**, pois o primário vai processando pedidos de vários clientes ao mesmo tempo, sendo que por cada pedido existe uma instância deste protocolo para representar a comunicação associada a ele. Portanto o primário não fica bloqueado à espera de receber as mensagens prepareok, ele vai recebendo-as ao longo tempo e quando tem f mensagens com um determinado *op-number* pode avançar para o próximo passo.

No estado ReplyPhase (linhas 24-28) o primário envia a mensagem reply para o cliente. Caso a opção de mensagem reply seja selecionada o protocolo termina. Caso ocorra um *timeout* a comunicação avança para o estado RetryPhase.

No estado RetryPhase (linhas 30-33) o cliente reenvia a mensagem request para todas as réplicas, que são representadas pelo conjunto rs **U** p. A opção de mensagem request leva a dois estados tal como no estado RequestPhase.

Por fim, no estado CommitPhase (linhas 35-39) o primário envia uma mensagem commit para as réplicas. A opção commit leva à terminação do protocolo e a opção **timeout** é igual à do estado PreparePhase.

Troca de primário. O protocolo global ViewChange, apresentado na figura 4.7 , representa o sub-protocolo que trata da eleição de um novo primário. O protocolo começa com a declaração das variáveis e dos papéis, neste caso apenas existe o papel Replica, pois o cliente não entra na comunicação deste protocolo.

```
1  type viewNumber, opNumber, commitNumber, latestViewNumber, replicaID = natural
2  type log = ...
3  role Replica
4
5  global protocol ViewChange(rps: Replica [2*f+1|f>0], xs: Replica [x| x>=f]){
6      ViewChangePhase:
7          message xs -> rps{
8              f:startviewchange(viewNumber, replicaID): DoViewChangePhase
9          }
10
11     DoViewChangePhase:
12         exists p:Replica. message rps -> p {
13             f+1: doviewchange(viewNumber, log, latestViewNumber, opNumber, commitNumber,
                   replicaID): StartViewPhase(p)
14         }
15
16     StartViewPhase(p:Replica):
17         message p -> rps {
18             startview(viewNumber, log, opNumber, commitNumber): end
19             timeout(): ViewChangePhase
20         }
21 }
```

Figura 4.7: Protocolo global da troca de primário.

Na assinatura do protocolo é declarado o conjunto rps, cujo papel é Replica e a sua aridade é dada pela expressão 2*f+1 e proposição f>0, portanto este conjunto representa

todas as réplicas. O segundo conjunto é identificado por rs, o seu papel é Replica e a sua aridade é dada pela expressão x e proposição x>=f, que indica que este conjunto consiste em pelo menos uma réplica. O conjunto xs representa as réplicas que iniciam o protocolo ViewChange.

No corpo do protocolo existem três estados que representam a comunicação global necessária para eleger outra réplica como primário. No estado ViewChangePhase (linhas 6-9) o conjunto xs envia uma mensagem startviewchange para o conjunto rps. Neste estado apenas é trocada a opção de mensagem startviewchange que leva ao estado DoViewChangePhase. As réplicas esperam por f mensagens desta opção para avançar na comunicação. Neste estado não existe a opção **timeout**, pois cada réplica tem uma *thread* dedicada ao protocolo de troca de primário, que espera por mensagens que iniciam este protocolo.

No estado DoViewChangePhase (linhas 11-14) é escolhida a réplica que será o novo primário, à qual é atribuído o identificador p e papel Replica. Depois cada réplica envia uma mensagem doviewchange para p. A opção de mensagem doviewchange direciona a comunicação para o estado StartViewPhase, no qual é passado como parâmetro o identificador p, pois este é necessário na troca de mensagens desse estado.

Antes da opção doviewchange é indicada a expressão f+1, assim o primário só pode avançar para o estado seguinte quando receber esse número de mensagens. Tal como ocorre com as réplicas o primário também tem uma *thread* especifica que fica à espera de mensagens doviewchange, portanto nesta opção não é necessário representar a opção **timeout**.

No estado StartViewPhase (linhas 16-20) p envia uma mensagem startview para o conjunto rps. Neste estado a opção de mensagem startview que leva à terminação do protocolo e a opção **timeout** direciona a comunicação para o estado ViewChangePhase, para iniciar uma nova troca de primário.

Recuperação após falha. O protocolo global Recovery, apresentado na figura 4.8, representa o sub-protocolo que trata da recuperação de réplicas que falharam. Neste protocolo apenas é declarado o papel Replica, tal como no protocolo anterior. Na assinatura do protocolo é declarado o participante que está a recuperar, através do identificador r, e o conjunto de réplicas excluindo r, que é definido pelo identificador rs.

No corpo do protocolo são definidos dois estados. No estado RecoveryPhase (linhas 6-9) r envia uma mensagem recovery para o conjunto rs. A opção de mensagem recovery leva ao estado RecoveryResponsePhase. Neste estado não existe a opção **timeout**, pois cada réplica tem uma *thread* dedicada para ajudar outras réplicas a recuperar.

No estado RecoveryResponsePhase (linhas 11-15) as réplicas respondem a r com uma mensagem recoveryresponse. A opção de mensagem recoveryresponse leva à terminação do protocolo. Neste caso r tem que esperar por pelo menos f+1 mensagens para que o protocolo termine e uma dessas mensagens tem que ser da réplica que é o primário. Caso

ocorra um *timeout* a comunicação é direcionada para o primeiro estado para a réplica voltar a tentar recuperar.

```
1  type viewNumber, opNumber, commitNumber, latestViewNumber, nonce, replicaID,  primaryID
       = natural
2  type log = ...
3  role Replica
4
5  global protocol Recovery(r: Replica, rs: Replica [2*f|f>0]){
6      RecoveryPhase:
7          message r -> rs{
8              recovery(replicaID, nonce): RecoveryResponsePhase
9          }
10
11     RecoveryResponsePhase:
12        message rs -> r {
13             f+1: recoveryresponse(viewNumber, nonce, log, opNumber, commitNumber,
                   replicaID): end
14             timeout(): RecoveryPhase
15         }
16 }
```

Figura 4.8: Protocolo global de recuperação após falha.

Transferência de estado. Na figura 4.9 é apresentado o protocolo global StateTransfer, que representa a transferência de estado, para o caso em que uma réplica se encontra atrasada em relação às outras. Na assinatura do protocolo apenas é declarado o participante que está atrasado, usando o identificador r e papel Replica.

```
1  Type viewNumber, opNumber, commitNumber, replicaID = natural
2  Type result, messageReceived, log = ...
3  Role Replica
4
5  global protocol StateTransfer(r: Replica){
6      GetStatePhase:
7          exists q:Replica. message r -> q {
8              getstate(viewNumber, opNumber, replicaID): StateResponsePhase(q)
9          }
10
11     StateResponsePhase(q:Replica):
12         message q -> r {
13             newstate(viewnumber, log, opNumber, commitNumber): end
14             timeout(): GetStatePhase
15         }
16 }
```

Figura 4.9: Protocolo global de transferência de estado.

O corpo do protocolo é constituído por dois estados. O estado GetStatePhase (linhas 6-9), começa com a escolha de uma das réplicas para interagir com a réplica atrasada, para a representar usa-se o identificador q. Depois r envia uma mensagem getstate para q. Quando esta mensagem é selecionada a comunicação avança para o estado StateResponsePhase , no qual é passado o identificador q como argumento. Neste estado não é definida a opção **timeout**, pois as réplicas têm uma *thread* especifica para ajudar réplicas atrasadas.

No estado StateResponsePhase (linhas 11-15) q envua uma mensagem newstate para r . Esta opção de mensagem leva à terminação do protocolo e a opção **timeout** faz a comunicação voltar ao primeiro estado para r pedir ajuda a outra réplica.

4.2.3 Protocolos locais

Nesta secção os protocolos globais que representam a comunicação global do protocolo VR são projetados para vários protocolos locais através das regras de tradução global-local apresentadas na secção 3.4. Assim obtêm-se os protocolos locais que representam a comunicação do ponto de vista de cada um dos participantes dos protocolos.

Operação normal. No protocolo global VR (figura 4.6) são declarados três identificadores na assinatura, o cliente, o primário e o conjunto de réplicas, portanto este protocolo origina três protocolos locais.

Protocolo local do cliente. O protocolo local VR_c que representa a comunicação do ponto de vista do cliente é apresentado na figura 4.10. No início do protocolo são declaradas as variáveis e os papéis tal como no protocolo global.

```
1  type opNumber, commitNumber, requestNumber, viewNumber, latestViewNumber = natural
2  type operation, clientID, replicaID, primaryID, nonce = natural
3  type result, messageReceived, log = ...
4  role Client, Replica
5
6  local protocol VR_c(c: Client, p: Replica, rs: Replica [2*f|f>0]){
7      RequestPhase:
8          send p {
9              request(operation, clientID, requestNumber): ReplyPhase
10         }
11
12     ReplyPhase:
13         receive p {
14             reply(viewNumber, requestNumber, result): end
15             timeout(): RetryPhase
16         }
17
18     RetryPhase:
19         multisend rs U p {
20             request(operation, clientID, requestNumber): ReplyPhase
21         }
22 }
```

Figura 4.10: Protocolo local da operação normal do cliente.

No corpo do protocolo apenas existem três estados, porque o participante c apenas participa em três estados do protocolo global VR. Nas trocas de mensagens c interage com p ou com o conjunto de todas as réplicas. Assim nas trocas com p é originada uma operação **send** ou **receive**, dependo se c se encontra antes ou após a seta de direção de mensagens no protocolo global. Na troca em que c interage com todas as réplicas (rs **U** p) a operação obtida é um **multisend**, porque c é o emissor. Neste protocolo a opção **timeout** apenas é representado no estado ReplyPhase, pois c está a receber uma mensagem.

Em alguns dos estados o nome dos estados para onde a comunicação avança é diferente do apresentado no protocolo global. No estado RequestPhase (linhas 7-10) a opção request pode levar aos estados PreparePhase ou ReplyPhase, mas no protocolo local do cliente não existe o estado PreparePhase. No entanto, se seguirmos os estados do protocolo global

seguindo a opção PreparePhase o participante c só volta a entrar no estado ReplyPhase. Assim no protocolo local de c é usado o nome do estado ReplyPhase em vez de PreparePhase. Mas ter ReplyPhase + ReplyPhase é o mesmo que ter apenas ReplyPhase, então no protocolo aparece só o nome do estado ReplyPhase no fim da opção request. O mesmo acontece no estado RetryPhase (linhas 18-21).

Protocolo local do primário. Na figura 4.11 é apresentado o protocolo local VR_p que representa a comunicação realizada no protocolo global VR do ponto de vista do primário.

```
1  type opNumber, commitNumber, requestNumber, viewNumber, latestViewNumber = natural
2  type operation, clientID, replicaID, primaryID, nonce = natural
3  type result, messageReceived, log = ...
4  role Client, Replica
5
6  local protocol VR_p(c: Client, p: Replica, rs: Replica [2*f|f>0]){
7      RequestPhase:
8          receive c {
9              request(operation, clientID, requestNumber): PreparePhase + ReplyPhase
10             timeout(): CommitPhase
11         }
12
13     PreparePhase:
14         multisend rs {
15             prepare(viewNumber, messageReceived, opNumber, commitNumber): PrepareOkPhase
16         }
17
18     PrepareOkPhase:
19         multireceive rs {
20             f: prepareok(viewNumber, opNumber, replicaID): ReplyPhase
21         }
22
23     ReplyPhase:
24         send c {
25             reply(viewNumber, requestNumber, result): end
26         }
27
28      RetryPhase:
29         receive c {
30              request(operation, clientID, requestNumber): PreparePhase + ReplyPhase
31         }
32
33     CommitPhase:
34         multisend rs {
35             commit(viewNumber, commitNumber): end
36         }
37 }
```

Figura 4.11: Protocolo local da operação normal do primário.

Neste protocolo existem seis estados tal como no protocolo global, porque o primário participa em todos os estados do protocolo global. Nas trocas de mensagens o primário interage com o cliente ou com o conjunto de réplicas. Assim quando este interage com o cliente as operações obtidas são o **send** ou **receive**, pois estas trocas são de um para um. Quando o primário interage com as réplicas são obtidas as operações **multisend** ou **multireceive**. A opção no fim da opção de mensagem **timeout** e as expressões que precedem as opções de mensagens apenas aparecem nas operações de receção.

A alternativa de estados na opção de mensagem request do estado RequestPhase (linhas 7-11) é traduzida através da regra nº8, que indica que os estados ficam separados pelo

símbolo + tal como no protocolo global.

Protocolo local das réplicas backup. Na figura 4.12 é apresentado o protocolo local VR_rs que representa a comunicação do protocolo global VR do ponto vista de cada uma das réplicas do conjunto rs.

```
1  type opNumber, commitNumber, requestNumber, viewNumber, latestViewNumber = natural
2  type operation, clientID, replicaID, primaryID, nonce = natural
3  type result, messageReceived, log = ...
4  role Client, Replica
5
6  local protocol VR_rs(c: Client, p: Replica, rs: Replica [2*f|f>0]){
7      PreparePhase:
8          receive p {
9              prepare(viewNumber, messageReceived, opNumber, commitNumber): PrepareOkPhase
10             timeout(r:Replica): ViewChange_xs(rs U p, r).ViewChangePhase
11         }
12
13     PrepareOkPhase:
14         send p {
15             prepareok(viewNumber, opNumber, replicaID): end
16         }
17
18     RetryPhase:
19         receive c {
20              request(operation, clientID, requestNumber): PreparePhase + end
21         }
22
23     CommitPhase:
24         receive p {
25             commit(viewNumber, commitNumber): end
26             timeout(r:Replica): ViewChange_xs(rs U p, r).ViewChangePhase
27         }
28 }
```

Figura 4.12: Protocolo local da operação normal das réplicas.

O corpo do protocolo local consiste em quatro estados, pois as réplicas do conjunto rs apenas participam em quatro estados do protocolo global. Neste protocolo cada réplica interage com o primário ou com o cliente, então na tradução global-local são obtidas as operações **send** ou **receive**.

A opção **timeout** no protocolo local fica um pouco diferente da do protocolo global. Como este protocolo local representa a comunicação de uma réplica esta opção vai ser só para essa réplica, portanto não se declara o conjunto xs. Neste caso identifica-se apenas a réplica a quem o protocolo local pertence, para tal é usado o identificador r e a expressão que antecede o **timeout** no protocolo global é removida. No fim desta opção a comunicação avança para o estado ViewChangePhase do protocolo ViewChange_xs, no qual é passado o identificador r em vez do xs, pois é esse que é declarado no **timeout** deste protocolo.

No estado prepareok (linhas 13-16) é alterado o estado para onde a comunicação continua, pois no protocolo global este estado é o ReplyPhase, que não faz parte do protocolo local de rs. Assim verificando o caminho das mensagens no estado ReplyPhase a comunicação termina em vez de avançar para outro estado, portanto no fim da opção prepareok aparece a palavra reservada **end**. O mesmo acontece no estado RetryPhase (linhas 18-21) no qual a opção request leva ao estado PreparePhase ou à terminação do protocolo.

Troca de primário. No protocolo global ViewChange (figura 4.7) são declarados os conjunto de réplicas rps e xs na assinatura. Para além disso ainda é declarado o novo primário através da palavra reservada **exists**. Assim o protocolo global ViewChange é projectado em três protocolos locais.

Protocolo de início da troca de primário. O protocolo local ViewChange_xs que representa a comunicação do protocolo global ViewChange do ponto de vista do conjunto de participantes xs é apresentado na figura 4.13.

```
1  type viewNumber, opNumber, commitNumber, latestViewNumber, replicaID = natural
2  type log = ...
3  role Replica
4
5  local protocol ViewChange_xs(rps: Replica [2*f+1|f>0], xs: Replica [x|x>=f]){
6      ViewChange:
7          multisend rps {
8              startviewchange(viewNumber, replicaID): end
9          }
10 }
```

Figura 4.13: Protocolo local de início de troca de primário.

O corpo do protocolo local apenas contém um estado, pois o conjunto de participantes xs apenas faz parte de um estado do protocolo global. No estado ViewChange cada réplica do conjunto xs envia mensagens startviewchange para o conjunto de participantes rps, portanto é obtido um **multisend**. A opção deste estado no protocolo global leva ao estado DoViewChangePhase, mas como este protocolo local não tem mais estados a comunicação termina. Neste estado não aparece a expressão f a anteceder a opção de mensagem.

Protocolo local das réplicas. Na figura 4.14 é apresentado o protocolo local ViewChange_rps que representa a comunicação do protocolo global ViewChange do ponto de vista de todas as réplicas.

```
1  type viewNumber, opNumber, commitNumber, latestViewNumber, replicaID = natural
2  type log = ...
3  role Replica
4
5  local protocol ViewChange_rps(rps: Replica [2*f+1|f>0], xs: Replica [x| x>=f]){
6      ViewChangePhase:
7          multireceive xs {
8              f: startviewchange(viewNumber, replicaID): DoViewChangePhase
9          }
10
11     DoViewChangePhase:
12         exists p:Replica. send p {
13             doviewchange(viewNumber, log, latestViewNumber, opNumber, commitNumber,
                   replicaID): StartViewPhase(p)
14         }
15
16     StartViewPhase(p:Replica):
17         receive p {
18             startview(viewNumber, log, opNumber, commitNumber): end
19             timeout(): ViewChangePhase
20         }
21 }
```

Figura 4.14: Protocolo local de troca de primário das réplicas.

O corpo do protocolo local é constituído por três estados, pois o conjunto de participantes rps participa em todas os estados do protocolo global. Neste protocolo as réplicas do conjunto rps interagem com o conjunto xs ou com o novo primário. Assim quando recebem mensagens do conjunto xs é obtido um **multireceive** e quando interagem com p obtem-se um **send** ou **receive**. A opção **timeout** e as expressões que precedem as opções apenas aparecem nos estados em que o conjunto rps é o recetor.

Protocolo local do novo primário. Na figura 4.15 é apresentado o protocolo local ViewChange_p que representa a comunicação do protocolo global ViewChange do ponto de vista do novo primário.

```
1  type viewNumber, opNumber, commitNumber, latestViewNumber, replicaID = natural
2  type log = ...
3  role Replica
4
5  local protocol ViewChange_p(rps: Replica [2*f+1|f>0], xs: Replica [x| x>=f]){
6      DoViewChangePhase:
7          multireceive rps {
8              f+1: doviewchange(viewNumber, log, latestViewNumber, opNumber, commitNumber,
                   replicaID): StartViewPhase
9          }
10
11     StartViewPhase:
12         multisend rps {
13             startview(viewNumber, log, opNumber, commitNumber): end
14         }
15 }
```

Figura 4.15: Protocolo local do novo primário.

O corpo do protocolo é constituído por dois estados. Em ambos os estados o primário interage com o conjunto rps, assim obtêm-se as operações **multisend** ou **multireceive** no protocolo local.

Recuperação após falha. Na assinatura do protocolo global Recovery (figura 4.8) é declarado o participante r, que é a réplica que está a recuperar, e um conjunto de participantes rs, que representa as restantes réplicas. Assim como são definidos dois identificadores na assinatura e nenhum no corpo do protocolo este protocolo global é projetado em dois protocolos locais.

Protocolo local de pedido de recuperação. O protocolo local Recovery_r, apresentado na figura 4.16, representa a comunicação do protocolo global Recovery do ponto de vista da réplica que está a recuperar.

O corpo do protocolo apresenta dois estados. Como em ambos os estados a réplica interagem com o conjunto de réplicas rs obtêm-se as operações **multisend** ou **multireceive** nas trocas de mensagens do protocolo local.

```
1  type viewNumber, opNumber, commitNumber, latestViewNumber, nonce, replicaID,  primaryID
       = natural
2  type log = ...
3  role Replica
4
5  local protocol Recovery_r(r: Replica, rs: Replica[2*f|f>0]){
6      RecoveryPhase:
7          multisend rs{
8              recovery(replicaID, nonce): RecoveryResponsePhase
9          }
10
11       RecoveryResponsePhase:
12         multireceive rs {
13             f+1: recoveryresponse(viewNumber, nonce, log, opNumber, commitNumber,
                   replicaID): end
14             timeout(): RecoveryPhase
15         }
16 }
```

Figura 4.16: Protocolo local da réplica que está a recuperar.

Protocolo local de resposta a pedidos de recuperação. Na figura 4.17 é apresentado o protocolo local Recovery_rs que representa a comunicação do protocolo global Recovery do ponto de vista das réplicas que atendem ao pedido de recuperação.

```
1  type viewNumber, opNumber, commitNumber, latestViewNumber, nonce, replicaID,  primaryID
       = natural
2  type log = ...
3  role Replica
4
5  local protocol Recovery_rs(r: Replica, rs: Replica[2*f|f>0]){
6      Recovery:
7          receive r{
8              recovery(replicaID, nonce): RecoveryResponse
9          }
10
11       RecoveryResponse:
12         send r {
13             recoveryresponse(viewNumber, nonce, log, opNumber, commitNumber, replicaID):
                   end
14         }
15 }
```

Figura 4.17: Protocolo local de resposta a pedidos de recuperação.

O corpo do protocolo apresenta dois estados, pois as réplicas participam em ambos os estados do protocolo global. Nestes estados as réplicas interagem com a réplica que está a recuperação, assim obtêm-se as operações **send** ou **receive** nas trocas de mensagens do protocolo local.

Transferência de estado. No protocolo global StateTransfer (figura 4.9) é declarado um participante na assinatura do protocolo e outro no corpo do protocolo. Na assinatura é declarada a réplica que está atrasada usando o identificador r, e no corpo do protocolo é declarada uma réplica para ajudar a que está atrasada, que é representada pelo identificador q. Assim o protocolo global é projetado em dois protocolos locais.

Protocolo local de pedido de transferência de estado. O protocolo local StateTransfer_r que representa a comunicação do protocolo global StateTransfer do ponto de vista da réplica que está atrasada é apresentado na figura 4.18.

```
1  type viewNumber, opNumber, commitNumber, replicaID = natural
2  type result, messageReceived, log = ...
3  role Replica
4
5  local protocol StateTransfer_r(r: Replica){
6      GetStatePhase:
7          exists q:Replica. send q {
8              getstate(viewnumber, opnumber, replicaID): StateResponsePhase(q)
9          }
10
11     StateResponsePhase(q:Replica):
12         receive q {
13             newstate(viewnumber, log, opnumber, commitnumber): end
14             timeout(): GetStatePhase
15         }
16 }
```

Figura 4.18: Protocolo local da réplica atrasada.

O corpo do protocolo local apresenta dois estados. Nestes estados r interage com a réplica q, que é declarada no estado GetStatePhase (linhas 6-9). Assim obtêm-se as operações **send** ou **receive** nas trocas de mensagens do protocolo local.

Protocolo local de resposta a pedidos de transferência de estado. Na figura 4.19 é apresentado o protocolo local StateTransfer_q que representa a comunicação do protocolo global StateTransfer do ponto de vista da réplica que atende ao pedido de atualização de estado.

```
1  type viewNumber, opNumber, commitNumber, replicaID = natural
2  type result, messageReceived, log = ...
3  role Replica
4
5  local protocol StateTransfer_q(r: Replica){
6      GetStatePhase:
7          receive r {
8              getstate(viewnumber, opnumber, replicaID): StateResponsePhase
9          }
10
11     StateResponsePhase:
12         send r {
13             newstate(viewnumber, log, opnumber, commitnumber): end
14         }
15 }
```

Figura 4.19: Protocolo local de resposta a pedidos de transferência de estado.

O corpo do protocolo apresenta dois estados, pois o participante q está envolvido em ambos os estados do protocolo global StateTransfer. Este protocolo é semelhante ao protocolo local anterior, mas nas trocas de mensagens o **send** é substituído pelo **receive** e vice-versa. Para além disso a opção **exists** também não faz parte deste protocolo local.

4.3 Considerações finais

Nesta secção foram apresentados dois protocolos mais complexos que o 2PC nos quais é possível ver como se aplicam algumas formas da linguagem que ainda não tinham sido utilizadas nos protocolos anteriores, nomeadamente a opção de seguir para dois estados diferentes que utiliza o símbolo +. Para além disso este capítulo permite perceber melhor como são aplicadas as regras para projeção global-local, apresenta situações em que um estado que aparece no protocolo global é alterado para um estado diferente no protocolo local seguindo o caminho das mensagens pelos estados.

Ao comparar os diversos protocolos locais conseguimos verificar que existe um emparelhamento entre as operações dos protocolos locais das duas partes comunicantes, por exemplo quando um participante envia uma mensagem num estado do seu protocolo local, o outro participante que está envolvido nessa troca recebe uma mensagem no estado correspondente do seu protocolo local. A juntar estas duas operações e comparando-as com a troca de mensagens representada no estado correspondente do protocolo global verifica-se que o participante que envia no seu protocolo local está antes da seta na troca de mensagens no protocolo global, e o participante que recebe no protocolo local aparece após a seta.

O caso de estudo continua no capítulo seguinte, no qual falamos sobre a nossa implementação dos protocolos 2PC e VR.

Capítulo 5

Implementação

Neste capítulo apresentamos uma pequena API que fornece quatro funções para envio e receção de mensagens num formato semelhantes ao dos protocolos locais. Depois mostramos excertos de código da nossa implementação do protocolo 2PC e VR, exemplificando como essa API pode ser utilizada. Usando as funções da nossa API torna-se fácil identificar que partes do código devem corresponder a cada estado dos protocolos locais. O que pode ser interessante para um trabalho futuro que mencionamos na conclusão.

Para criar a API e implementar os protocolos 2PC e VR é usada a linguagem *Erlang* [1]. O *Erlang* é uma linguagem funcional, com suporte incorporado para concorrência, distribuição e tolerância a falhas. Esta linguagem facilita a programação paralela, pois os processos apenas interagem por passagem de mensagens e não partilham memória, assim não existem os problemas de sincronização, *locks* e a possibilidade da memória partilhada ser corrompida.

Nas linguagens funcionais, as funções e operações da linguagem são semelhantes a cálculos matemáticos, no sentido em que as funções recebem um *input* e geram um resultado. O paradigma da programação funcional significa que os pedaços de código geram valores consistentes para os mesmos valores de *input*. Assim é fácil prever o resultado da função ou do programa, logo também é facilitada a depuração e analise do programa.

A linguagem *Erlang* apresenta um modelo de programação simples que é fácil de aprender. Para além disso como esta linguagem usa *pattern matching* os programas são mais pequenos do que os criados usando as linguagens *C* ou *Java*. Com todas estas caraterísticas o *Erlang* oferece um bom modelo para construir aplicações distribuídas, escaláveis e de alto desempenho.

5.1 API de comunicação para Erlang

A nossa API oferece quatro funções que representam as operações de comunicação apresentadas nos protocolos locais, ou seja, `send`, `multisend`, `receive` e `multireceive`. Assim é possível construir programas que estejam de acordo com os protocolos locais.

Em *Erlang* cada processo tem um *pid* (*Process Identifier*) que o distingue dos outros

processos e que é utilizado no envio das mensagens como destino. O código em *Erlang* é estruturado em módulos, sendo que em cada módulo são declaradas várias funções. As funções da API de comunicação foram declaradas no módulo *message*, portanto sempre que é necessário invocá-las noutro módulo é usado o nome do módulo *message* separado por dois pontos do nome da função escolhida e os seus parâmetros, como iremos mostrar na secção da implementação.

A função `send` representa a operação de comunicação *send* da sintaxe de protocolos locais e consiste no envio de uma mensagem de um participante para outro participante. Esta função recebe dois parâmetros, o primeiro parâmetro representado pela variável `Message` consiste na mensagem que o participante quer enviar. As mensagens na nossa implementação consistem num tuplo `{Label,Params}`, em que a `Label` é a etiqueta da mensagem e a variável `Params` consiste num tuplo $\{Param_1,..,Param_n\}$, no qual são passados os parâmetros da mensagem. Assim o formato da mensagem é $\{Label,\{Param_1,..,Param_n\}\}$, caso a mensagem não tenha parâmetros o seu formato é `{Label,{}}`.

O segundo parâmetro `Pid` é o identificador do processo para quem a mensagem será enviada, que é necessário para fazer o envio da mensagem. Esta função não tem nada de especial, apenas recebe os parâmetros e usa a função de envio do *Erlang*, que é representada pelo símbolo !. Assim quando esta função é executada chama-se a função de envio básica do *Erlang* e o envio está concluído.

```
send(Message, Pid)
```

A função `multisend` representa a operação de comunicação *multisend* da sintaxe de protocolos locais e consiste no envio de uma mensagem de um participante para um conjunto de participantes. Esta função também apresenta dois parâmetros, a variável `Message`, que é a mensagem que será enviada e a variável `Pids`, que consiste na lista de identificadores dos processos para quem a mensagem deve ser enviada, cujo o formato é $[Pid_1,..,Pid_n]$.

Nesta função recorre-se à função de envio do *Erlang* para fazer o envio da mensagem para cada um dos participantes. Mas ao contrário da função `send`, após se chamar a função de envio volta-se a chamar a função `multisend`, que é executada de forma recursiva até que se tenha enviado uma mensagem para todos os participantes que se encontram na lista `Pids`. Assim em cada passo da recursão é enviada uma mensagem para cada processo, até que a lista termine.

```
multisend(Message, Pids)
```

A função `recv` representa a operação de comunicação *receive* da sintaxe de protocolos locais e consiste na receção de uma mensagem. Optámos pelo nome `recv`, pois a função de receção básica do *Erlang* tem o nome *receive*. A função `recv` recebe dois parâmetros, o primeiro parâmetro representado pela variável `LOptions` consiste na lista de etiquetas

de mensagens que podem ser recebidas. A variável `LOptions` é um lista de tuplos, sendo que em cada tuplo se encontra o nome de uma etiqueta de mensagem, assim o formato da lista é `[{Label`$_1$`},..,{Label`$_n$`}]`. Nós optámos por criar um lista de tuplos em vez de ter apenas um lista com as etiquetas, pois em *Erlang* as funções de listas são orientadas para tuplos. Assim a pesquisa de elementos e outras operações podem ser feitas usando as funções da API do *Erlang*.

O segundo parâmetro é representado pela variável `Timeout` e consiste no tempo em segundos que o participante deseja esperar por uma mensagem que tenha uma das etiquetas da lista `LOptions`. Se o participante deve ficar bloqueado até receber uma dessas mensagens o valor da variável `Timeout` é zero. Caso contrário deve ser indicado um número a partir de um, para definir o tempo que o participante tem que esperar até que ocorra um *timeout*.

Esta função recorre à função de receção básica do *Erlang* para receber uma mensagem que apresente uma das etiquetas indicadas na lista de etiquetas. Nessa função são indicados os formatos das mensagens que pretendem receber, para tal usa-se um tuplo `{Label,Params}`, no elemento `Label` é indicada um das etiquetas que se encontra na lista de etiquetas, assim cada uma das etiquetas é representada por um deste tuplos. Após cada tuplo é indicado o que deve acontecer quando se recebe uma mensagem com a etiqueta indicada, neste caso em cada uma das opções a função termina e é devolvida a mensagem recebida.

Entretanto pode ocorrer um *timeout* se o tempo definido na variável `Timeout` expirar, nesse caso a função termina e retorna o tuplo `{timeout,{}}`, para o utilizador saber que ocorreu um *timeout*.

```
recv(LOptions, Timeout)
```

A função `multireceive` representa a operação de comunicação *multireceive* da sintaxe de protocolos locais e consiste na receção de várias mensagens. Esta função também recebe dois parâmetros, o primeiro parâmetro representado pela variável `LOptions` consiste na lista de etiquetas de mensagens que o participante pode receber. O segundo parâmetro é representado pela variável `Timeout` e representa o tempo que o processo fica à esperar de receber as mensagens, tal como foi referido na função `recv`.

A lista de etiquetas `LOptions` pode apresentar dois formatos, pode ser uma lista de tuplos com dois ou três elementos. No caso de ser constituída por tuplos com dois elementos, o primeiro elemento é etiqueta da mensagem e o segundo é o número de mensagens que devem ser recebidas com essa etiqueta.

Por vezes pode ser necessário verificar alguma propriedade sobre os parâmetros das mensagens, como por exemplo quando se espera por um determinado número mensagens que tenham a mesma etiqueta e que tenham todas o mesmo segundo parâmetro. Para estes casos a lista de etiquetas é constituida por tuplos de três elementos, em que o terceiro elemento consiste no tuplo `{Module,Function}`, contendo o nome do módulo e o nome

da função a executar para filtrar as mensagens recebidas verificando uma determinada propriedade nos seus parâmetros. Esta função recebe a lista de mensagens recebidas até ao momento na função `multireceive`, verifica cada mensagem segundo a propriedade que procura e devolve a lista de mensagens para as quais a propriedade é verdadeira.

A função `multireceive` é um pouco mais complicada que a `recv`, ela usa o `receive` fornecido na API do *Erlang* e quando recebe uma mensagem cuja etiqueta corresponde a uma das da lista de etiquetas guarda-a numa lista de mensagens. Depois é chamada uma função que analisa se as mensagens que estão nessa lista já são as necessárias ou não. No caso em que a lista de etiquetas é constituída por tuplos de dois elementos é usada uma lista de contadores constituída por tuplos, um para cada uma das possíveis etiquetas de mensagens. O segundo elemento de cada tuplo é o número de mensagens já recebidas com essa etiqueta, assim esta lista apresenta o seguinte formato `[{Label`$_1$`,NReceived`$_1$`},..,{Label`$_n$`,NReceived`$_n$`}]`. Então quando recebe uma mensagem para além de a guardar na lista de mensagens recebidas também incrementa o número de mensagens recebidas que têm essa etiqueta no tuplo correspondente da lista de contadores.

Sempre que se encontra uma correspondência é verificado se o número de mensagens necessário para essa etiqueta corresponde ao número de mensagens recebidas, caso isso aconteça a função termina e devolve a lista de mensagens recebidas, que é limpa de outras mensagens que tenham etiquetas diferentes da opção que foi selecionada, mas que foram guardadas durante o processo de receção. Caso o número de mensagens necessárias ainda não tenha sido atingido continua a recursão e a função `multireceive` é novamente executada.

Quando a lista de etiquetas é constituída por tuplos de três elementos o processo é um pouco diferente. Primeiro a mensagem é adicionada à lista de mensagens recebidas e depois é chamada a função indicada no terceiro elemento do tuplo, que devolve a lista de mensagens que apresentam a mesma etiqueta que a última mensagem recebida e têm uma determinada propriedade. Caso a lista de mensagens devolvida pela função tenha o número de mensagens necessárias a função termina e devolve essa lista de mensagens, caso contrário volta-se a executar a função `multireceive` até receber as mensagens necessárias.

Entretanto pode ocorrer um *timeout* caso o participante não receba o número de mensagens necessárias de nenhuma das opções, neste caso a função retorna o tuplo `{timeout,{}}` para o utilizador saber que o tempo expirou sem que tenham sido recebidas as mensagens necessárias.

```
multireceive(LOptions, Timeout)
```

As funções de receção não são genéricas, pois como queremos utilizar a função *receive* do *Erlang* para fazer o *pattern matching* das mensagens temos que criar uma função para cada número de tuplos que a lista de etiquetas pode ter. Neste caso as funções `recv` e `multireceive` estão preparadas para receber uma lista de etiquetas com um máximo

de três tuplos, pois estas são as funções que são necessárias para a nossa implementação. Caso seja necessário usar a nossa API para receber uma lista com mais opções dos que as disponíveis teria que se criar uma extensão para a API com funções que cobram os casos necessários.

O código da API encontra-se no apêndice A.

5.2 Implementação dos Protocolos

Como foi referido anteriormente cada protocolo local tem um programa individual que representa a ação do processo a quem o protocolo local pertence. Assim na implementação é criado um módulo para cada protocolo local, que usa as funções da API para representar as trocas de mensagens do protocolo local. Por vezes pode-se dividir a comunicação de um protocolo por mais módulos, se for necessário. Como os módulos seguem os protocolos locais é possível verificar que partes do código correspondem a que estado e se a ordem dos estados do protocolo é respeitada ou não.

5.2.1 Protocolo Two-Phase Commit

Para implementar o 2PC criámos cinco módulos, que se encontram representados na figura 5.1. Nos diagramas de módulos são usadas setas para representar que um módulo usa funções de outro módulo. O módulo `twophasecommit_c` consiste na execução normal do coordenador, assim as trocas de mesagens deste módulo correspondem às trocas descritas no protocolo local *TwoPhaseCommit_c*. Os outros quatro módulos consistem na ação de um participante. O módulo `central` é o ponto central do participante, onde este recebe todas as mensagens que vêm do exterior, provenientes do coordenador ou de outro participante. Quando o processo responsável por este módulo recebe uma mensagem reencaminha-a para o módulo a que esta diz respeito, onde esta vai ser processada por outro processo. Para além disso, o módulo `central` também guarda o estado do participante, que corresponde aos estados dos protocolos locais do participante.

Os módulos `twophasecommit_p`, `decision_p` e `decision_q` como os seus nomes indicam correspondem aos protocolos locais *TwoPhaseCommit_ps*, *Decision_p* e *Decision_q*, respetivamente. Cada participante tem três processos que executam ao mesmo tempo, um que executa o módulo `central` para receber mensagens do exterior, um para o módulo `twophasecommit_p` que é responsável pela execução normal do participante e outro para o módulo `decision_q` que responde a pedidos de decisão. O coordenador apenas tem um processoque executa o módulo `twophasecommit_c`.

De seguida iremos mostrar excertos de cada módulo evidenciando a correspondência entre as operações de comunicação apresentadas no código com as dos protocolos locais correspondentes. O código completo de cada módulo encontra-se em anexo.

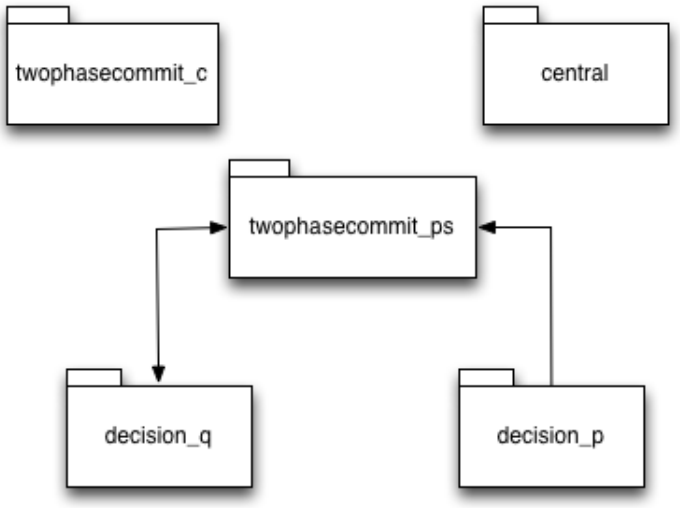

Figura 5.1: Diagramas de módulos que implementam o protocolo 2PC.

Módulo twophasecommit_c. No módulo `twophasecommit_c` (figura B.1) existe uma função `coordinator`, responsável pela ação normal do coordenador. Esta função recebe o estado do coordenador e a lista de *pids* dos participantes. O estado é inicialmente `phaseOne`, tal como no protocolo local *TwoPhaseCommit_c* (figura 3.9), e depois é alterado consoante se avança na ação. A função `coordinator` encontra-se dividida em várias partes para dar uma ideia de divisão das trocas de mensagens por estados, ficando semelhante à estrutura dos protocolos.

```
...
coordinator(phaseOne, LPids) ->
    message:multisend({vote_request,{}}, LPids),
    coordinator(waitReady, LPids);
...
coordinator(waitReady, LPids) ->
    LMessages = message:multireceive([{vote_commit,length(LPids)},{
       vote_abort,1}], 10),
...
coordinator(twoPhaseCommit, LPids) ->
    message:multisend({global_commit,{}}, LPids);
...
coordinator(twoPhaseAbort, LPids) ->
    message:multisend({global_abort,{}}, LPids).
```

Na função `coordinator` que recebe o argumento `phaseOne` é chamada a função `multisend` da API, que recebe como argumentos a mensagem que se vai enviar, que neste caso tem a etiqueta `vote_request` e não tem argumentos, e a lista de *pids* de todos os participantes. Após a função `multisend`, é invocada a função `coordinator` com `waitReady` como primeiro argumento.

Na função `coordinator` que recebe o estado `waitReady` é chamada a função `multireceive` da API, que recebe a lista de opções de mensagens, o primeiro tuplo con-

siste na etiqueta `vote_commit` e no número de participantes, dado pelo tamanho da lista de *pids*, o segundo tuplo consiste na etiqueta `vote_abort` e no número um. O segundo argumento da função `multireceive` consiste no tempo que o participante espera antes de fazer *timeout*, neste caso esse valor é 10 segundos.

A função `multireceive` retorna uma lista com as mensagens recebidas da opção selecionada, no caso de ocorrer um *timeout* retorna uma lista com o tuplo `{timeout,{}}`. A lista com as mensagens recebidas é analisada e caso as mensagens tenham a etiqueta `vote_commit` é chamada a função `coordinator` com o estado `phaseTwoCommit` como argumento. Caso a lista tenha uma mensagem com a etiqueta `vote_abort` ou o tuplo que indica que ocorreu *timeout* é chamada a função `coordinator` com o estado `phaseTwoAbort` como argumento. Na implementação existe esta divisão do estado *PhaseTwo*, pois é necessário representar o envio de cada uma das opções de mensagens desse estado. Assim consoante as mensagens recebidas segue-se para partes da função diferentes nas quais são enviadas mensagens diferentes.

Na função `coordinator` que recebe o estado `phaseTwoCommit` é chamada a função `multisend` da API. Esta função recebe a mensagem que será enviada, que tem a etiqueta `global_commit` e não tem argumentos, e a lista com os *pids* de todos os participantes.

A função `coordinator` que recebe o estado `phaseTwoAbort` como argumento corresponde ao outro lado do estado *PhaseTwo*, que seleciona a opção *global_abort*. Neste excerto chama-se a função `multisend` da API, mas neste caso o primeiro argumento tem a etiqueta `global_abort`. Em ambas as funções o processo termina após a chamada à função `multisend`, tal como acontece no protocolo local.

Comparando o protocolo local *TwoPhaseCommit_c* com este módulo é fácil verificar que existem uma correspondência em termos de comunicação. Sempre que uma função da API é chamada existe uma troca de mensagens correspondente no protocolo local, que apresenta a mesma operação de envio ou receção, as mesmas opções de mensagens e a opção *timeout* se aplicável. Para além disso essa função leva à mesma troca de mensagens que no protocolo local. Na implementação as receções são representados tal como nos protocolos locais, com todas as opções de mensagens agrupadas na mesma função, indicando que nesse pode pode ser recebida qualquer uma dessas funções. No caso dos envios as opções de mensagens não se encontram agrupadas numa lista pois tem que haver um escolha das opções de mensagens, para ser claro qual a mensagem que será passada pela função de envio. No entanto, se se observarem as várias funções de envio é possível perceber que em conjuntos elas formam a lista de as opções da troca de mensagens apresentada no protocolo local.

Com a passagem pelas diferentes funções é possível verificar que a ordem das mensagens do protocolo local é preservada, por exemplo após o envio da mensagem `vote_request` existe uma receção das mensagens de voto, como ocorre no protocolo local. Para além disso no ponto em que o protocolo local termina o processo também

termina.

Módulo twophasecommit_p O módulo `twophasecommit_p` (figura B.3) consiste na ação normal do participante. Neste módulo existe a função `participant`, que tem como argumento o estado do participante, que corresponde aos estados dos protocolos locais. O estado inicial é `phaseOne`, como no protocolo local *twophasecommit_ps* (figura 3.10).

Na função `participant` que recebe o estado `phaseOne` como argumento é chamada a função `recv` da API. Esta função recebe a lista com as etiquetas das mensagens que podem ser recebidas neste estado e define um *timeout* de 10 segundos.

A função `recv` devolve a mensagem recebida ou `{timeout,{}}`. Quando o participante recebe a mensagem `vote_request` o estado é alterado para `waitReadyVote`, caso ocorra um *timeout* o estado é alterado para `waitReadyAbort`. No protocolo local ambas as opções seguem para o estado *WaitReady*, mas na implementação é necessário fazer a separação entre o estado em que o participante escolhe um voto e o estado em que o participante aborta porque não recebeu nenhuma mensagem.

```
...
participant(phaseOne, Central, Coordinator, LParticipants) ->
    {Label,_} = message:recv([{vote_request}], 10),
...
participant(waitReadyVote, Central, Coordinator, LParticipants) ->
    Vote = chose([vote_commit,vote_abort]),
    message:send({Vote,{}}, Coordinator),
...
participant(waitReadyAbort, Central, Coordinator, LParticipants) ->
    Label = message:send({vote_abort,{}}, Coordinator),
...
participant(phaseTwo, Central, Coordinator, LParticipants) ->
    {Label,_} = message:recv([{global_commit},{global_abort}], 10),
...
```

Na função `participant` que recebe o estado `waitReadyVote` como argumento o participante escolhe o seu voto, que pode ser `vote_commit` ou `vote_abort`. Depois do voto estar escolhido é chamada a função `send` da API para fazer o envio da mensagem para o coordenador. Esta função recebe a mensagem a ser enviada e o *pid* do coordenador. De seguida o estado é alterado para `phaseTwo`.

Na função `participant` que recebe o estado `waitReadyAbort` como argumento, é chamada a função `send` da API para fazer o envio da mensagem para o coordenador. Esta função recebe a mensagem a ser enviada, que neste caso tem a etiqueta `vote_abort` e não tem argumentos, e o *pid* do coordenador. De seguida o estado é alterado para `phaseTwo`. Apesar de haver esta divisão entre a parte da função que representa o participante que vota e a que representa o participante que aborta, ambas avançam para a parte da função que recebe o estado `phaseTwo`, como ocorre no protocolo local.

Na função `participant` que recebe o estado `phaseTwo` como argumento, é chamada a função `recv` para receber a decisão do coordenador. No primeiro argumento da função é

passada a lista de mensagens que podem ser recebidas, identificadas pelas suas etiquetas. O segundo argumento identifica quantos segundos o participante espera até ocorrer um *timeout*, que neste caso são 10 segundos.

A função `recv` retorna a mensagem recebida ou a expressão `{timeout,{}}`. Caso a mensagem recebida tenha a etiqueta `global_commit` o estado é alterado para `commit` e o processo termina. Se a etiqueta da mensagem for `global_abort` o estado é alterado para `abort` e o processo também termina. Caso ocorra um *timeout* é chamada a função `decision` do módulo `decision_p` para pedir a decisão a outro participante.

Neste módulo cada parte da função `participant` corresponde a um estado do protocolo local, representando a troca de mensagens desse estado tal como descrita no protocolo.

Módulo decision_p No módulo `decision_p` (figura B.4) existe a função `decision` que consiste no pedido de decisão a outro participante e na receção da resposta a esse pedido. Neste caso o envio e a receção da decisão são representados na mesma parte da função, em vez de os estar a separar em várias estados como acontece no protocolo local, pois o que interessa é a ordem das operações de comunicação e não o nome do estado para onde a comunicação segue.

```
...
decision(Central, Coordinator, LParticipants) ->
    Pid = chose(LParticipants),
    message:send({decision_request,{self()}}, Pid),

    {Label2,_} = message:recv([{dec_global_commit},{dec_global_abort},{
        no_decision}], 10),
...
```

Esta função começa com a escolha aleatória de um participante da lista de *pids* de participantes, para tal é usada a função `chose` que corresponde ao *exists* do protocolo local *Decision_p* (figura 3.11). Com o participante escolhido, é chamada a função `send` da API que corresponde ao estado *AnkDecision* do protocolo local. Esta função recebe a mensagem que vai ser enviada, que tem a etiqueta `decision_request` e como argumento da mensagem recebe a função `self()` que retorna o *pid* do participante. No protocolo local esta opção de mensagem não tem parâmetros, mas na implementação é necessário passar um parâmetro com o *pid* do emissor para que o processo que recebe o pedido consiga enviar a resposta. O segundo argumento da função é o *pid* do participante escolhido através da função `chose`, que corresponde ao participante *q* criado no protocolo local.

De seguida é chamada a função `recv` da API que corresponde à troca de mensagens do estado *AnswerDecision*. Esta função recebe dois argumentos, a lista com as etiquetas das mensagens que podem ser recebidas e o tempo em segundos que se espera antes de ocorrer um *timeout*, que neste caso é 10. Caso a mensagem devolvida tenha a etiqueta `dec_global_commit` o estado do participante é alterado para `commit`, caso a etiqueta seja

`dec_global_abort` é alterado para `abort`, em ambos os casos o processo termina. Caso a etiqueta seja `no_decision` ou tenha ocorrido um *timeout* é chamada novamente a função `decision` para pedir a decisão a outro participante.

Ambas as funções da API usadas neste módulo correspondem às operações das trocas de mensagens descritas no protocolo local `Decision_p`, estas recebem os mesmos argumentos e aparecem na mesma ordem que no protocolo.

Módulo decision_q O módulo `decision_q` (figura B.5) consiste na receção de pedidos de outros participantes. A comunicação apresentada neste módulo corresponde à do protocolo local *Decision_q* (figura 3.9). Este módulo apenas tem uma função, cujo nome é `decision`, a qual começa com a chamada à função `recv` da API para receber o pedido de decisão. A função `recv` recebe dois argumentos, a lista com as etiquetas de mensagens que pode receber, neste caso só tem a etiqueta `decision_request`, o segundo argumento indica que neste caso o participante nunca faz *timeout*, pois o valor deste argumento é zero, assim este processo fica bloqueado até receber uma mensagem com a etiqueta `decision_request`.

```
...
decision(Central) ->
    {Label, {Pid}} = message:recv([{decision_request}], 0),
...
    if
        State == commit ->
            message:send({dec_global_commit,{}}, Pid);
        State == abort ->
            message:send({dec_global_abort,{}}, Pid);
        State == phaseOne ->
            message:send({dec_global_abort,{}}, Pid),
            twophasecommit_ps:set_state(Central,abort);
        State == phaseTwo ->
            message:send({no_decision,{}}, Pid)
...
```

A função `recv` retorna a mensagem recebida, que tem como argumento o *pid* do participante que enviou o pedido, que o participante irá utilizar para enviar a resposta. De seguida é verificado o estado do processo e consoante o estado é chamada a função `send` da API para enviar a decisão para o participante.

Comparando a função `recv` do excerto de código com o estado *AskDecision* do protocolo local *Decision_q*, pode-se observar que em ambos existe uma receção da mensagem *decision_request*. No estado *AskDecision* não existe a opção *timeout* e na função `recv` também é indicado que não existe *timeout* pelo segundo argumento da função, que consiste no número zero. Tanto no código como no protocolo esta função é seguida de um envio.

Na função `decision` existem varias chamadas à função `send` da API, comparando cada uma delas com o estado *AnswerDecision*, pode-se verificar que as funções `send`

correspondem à operação *send p* do protocolo local. No protocolo local existem três opções de mensagens que podem ser enviadas, cada uma das funções `send` corresponde à seleção de uma dessas opções. No protocolo local as opções *dec_global_commit* e *dec_global_abort* levam à terminação do protocolo e a opção *no_decision* leva o protocolo para o estado *AskDecision* para poder atender a outros pedidos de decisão.

Na implementação independentemente da escolha de mensagem enviada é chamada a função `decision`, para poder atender a outros pedidos dos participantes. O protocolo termina, pois cada instância do protocolo local consiste em apenas uma execução da comunicação apresentada, ou seja, o protocolo local *decision_p* consiste apenas no pedido de decisão do participante *p* ao participante *q*, o pedido dos outros participantes a *q* ou a outro participante são representados por outra instância.

Como é possível verificar a comunicação apresentada na implementação corresponde à apresentada nos protocolos locais. Ao seguir o caminho que a comunicação toma na execução é possível verificar que esta está de acordo com os estados dos protocolos locais. Em cada ponto da comunicação conseguimos ver qual foi a opção de mensagem selecionada nesse ponto e seguir por ai a comunicação até ao fim do protocolo.

A implementação por vezes inclui passos extra que não estão no protocolo, pois pode ser necessário fazer outras operações de comunicação que não estão diretamente relacionadas com a ação normal do sistema. Por exemplo neste programa o processo responsável pelo módulo `central` troca mensagens com os módulos `twophasecommit_p` e `decision_q` para ir atualizando o estado do participante. Estas trocas de mensagens não precisam de fazer parte do protocolo, pois estão relacionadas com a forma que o programador decide implementar o seu programa. Outro exemplo é o argumento que é passado na mensagem `decision_request` que só faz sentido aparecer na implementação do sistema e não no protocolo, pois este é um passo extra que é necessário devido à forma como decidimos implementar o programa.

5.2.2 Protocolo ViewStamped Replication

O protocolo VR é muito mais complexo que o 2PC, pois os participantes têm que guardar um estado com muitas variáveis, as réplicas podem executar dois tipos de código diferentes, dependendo se são o primário ou uma réplica *backup*, e o primário recebe pedidos de vários clientes ao mesmo tempo que comunica com as réplicas para as informação dos pedidos. Ao contrário do 2PC em que apenas é executada uma instância do protocolo de cada vez, no VR existem várias instâncias a serem executadas ao mesmo tempo para clientes diferentes, o que torna mais complicado fazer a correspondência entre o protocolo local e o programa.

Para implementar o protocolo VR foi criado um módulo para cada protocolo local e quatro estados adicionais, tendo assim um total de quatorze módulos. A ação do cliente é representada em apenas um módulo, cujo nome é `vr_c`, assim o cliente apenas

tem um processo a executar a sua ação. Os restantes módulos representam a ação das réplicas. Nesta implementação também existe um módulo `central` que é o ponto central das réplicas, onde é recebida toda a comunicação proveniente de clientes e outras réplicas. Este módulo apresenta uma função específica para o primário e outra para as réplicas, pois estes participantes recebem mensagens diferentes que são tratadas para módulos diferentes.

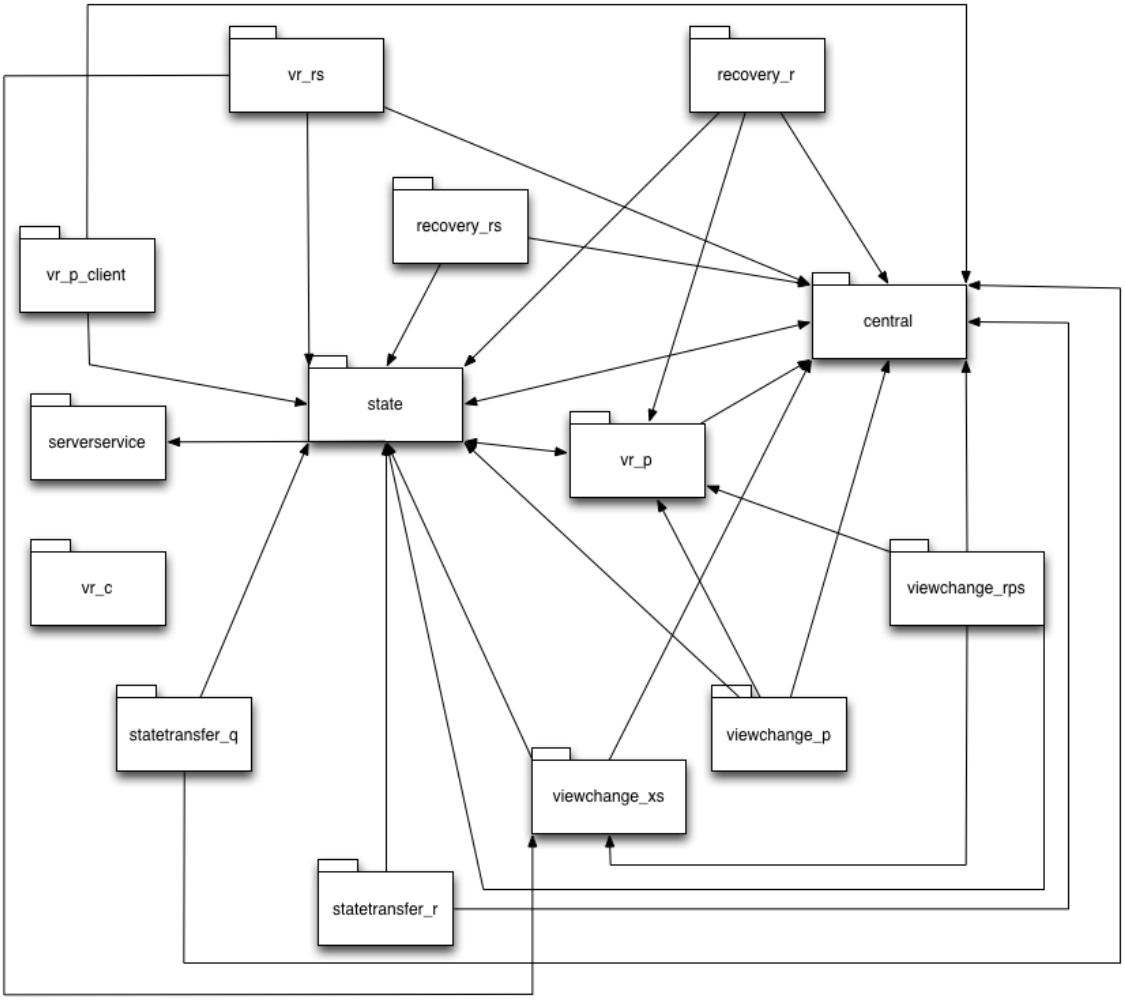

Figura 5.2: Diagramas de módulos que implementam o protocolo VR.

A ação normal do primário está dividida por dois módulos, o `vr_p_client`, no qual o primário recebe os pedidos dos clientes e após receber um pedido envia mensagens `prepare` para as réplicas, e o módulo `vr_p` que espera por mensagens `prepareok` das réplicas e envia as respostas para o cliente. O módulo `vr_rs` consiste na ação normal das réplicas *backup*, em que elas recebem mensagens `prepare` do primário e respondem com uma mensagem `prepareok`.

O módulo `viewchange_xs` dá inicio à troca de primário quando ocorre um *timeout*

na réplica, para tal envia mensagens `startviewchang` para as outras réplicas. O módulo `viewchange_rs` consiste na troca de primário, neste módulo as réplicas ficam à espera de recebem mensagens `startviewchange` e quando recebem as necessárias continuam com o processo para eleger o novo primário. No módulo `viewchange_p` as réplicas esperam por mensagens `doviewchange`, que irão receber caso sejam o novo primário, quando o novo primário recebe as mensagens necessárias envia mensagens `startview` para todas as réplicas e depois volta a ficar bloqueado à espera de mensagens `doviewchange`.

O módulo `recovery_r` consiste na ação para a réplica conseguir recuperar e o módulo `recovery_rs` consiste na ação para responder a pedidos de recuperação. No módulo `recovery_rs` as réplicas ficam bloqueadas à espera de pedidos de recuperação, quando recebem uma enviam a resposta e depois voltam a ficar bloqueadas à espera de mais pedidos de recuperação.

O módulo `statetransfer_r` consiste no pedido de atualização de estado, no caso em que a réplica se encontra atrasada e o módulo `statetransfer_q` consiste na resposta a pedidos de atualização de estado. No módulo `statetransfer_q` as réplicas ficam bloqueadas à espera de uma mensagem `getstate`, quando a recebem respondem com uma mensagem `newstate`. Após ajudarem a réplica ficam novamente bloqueadas à espera de atender a mais pedidos de atualização de estado.

O módulo `state` apresenta diversas funções que tratam da criação e alteração do estado das réplicas. Por fim, o módulo `serverservice` consiste no serviço a que os clientes pretendem aceder, neste caso o nosso módulo apenas imprime uma frase no ecrã do terminal que diz que executou a operação número x para o cliente y. Num sistema real este serviço seria algo mais complexo.

O cliente apenas tem um processo a executar o módulo `vr_c`. Todas as réplicas têm 6 ou 7 processos a ser executados simultaneamente, dependendo se são o primário ou não, sendo que têm um processo para cada um dos seguintes módulos: `central`, `viewchange_rs`, `viewchange_p`, `recovery_rs` e `statetransfer_q`. Para além dos processos referidos acima o primário tem ainda um processo a executar o módulo `vr_p_client` e outro a executar o módulo `vr_p`, enquanto que as réplicas *backup* têm um processo para executar o módulo `vr_rs`.

De seguida iremos mostrar alguns excertos que mostram situações um pouco diferentes das usadas na implementação do 2PC, para explicar como são aplicados na implementação. No módulo `vr_p` é utilizado a função `multireceive` da API para receber `f` mensagens `prepareok`, mas neste caso a lista de etiquetas de mensagens passadas na função consistem em tuplos de três elementos. São usadas estes tuplos, porque é necessário receber mensagens `prepareok` que tenham o mesmo `OpNumber`, visto que as mensagens têm que pertencer ao mesmo pedido para se poder enviar a resposta para o cliente.

Para filtrar as mensagens que são recebidas no `multireceive` é usada a função

`filter_prepareok` que verifica se as mensagens da lista têm a etiqueta `prepareok` e que o segundo elemento dos parâmetr,o que corresponde ao `OpNumber`, é igual ao da mensagem que acabou de receber. Após receber as mensagens necessárias avança para a função `primary` com argumento `replyPhase`, onde é enviada a mensagem `reply` para esse pedido, caso os pedidos anteriores já tenham sido respondidos. Este `multireceive` corresponde ao estado *PrepareOkPhase* do protocolo local *VR_p*, no qual é definido uma operação de receção *multireceive*, que tem apenas a opção de mensagem *prepareok* que leva ao estado *ReplyPhase*, tal como na implementação.

No início do módulo `vr_rs` é chamada a função `recv` da API, que recebe uma lista com as etiquetas `prepare` e `commit` e apresenta um *timeout* de 15 segundos.

```
...
primary(prepareOkPhase, Central) ->
    ...
    LMessages = message:multireceive([{prepareok,Necessary,{vr_p,
       filter_prepareok}}], 0),
...
```

Quando o processo recebe uma mensagem `prepare` é chamada a função `replica` que recebe o argumento `preparePhase`, na qual é verificado se o pedido tem o `OpNumber` seguinte ao último recebido. Caso isso aconteça é chamada a função `replica` que recebe o argumento `prepareOkPhase`, na qual é enviada uma mensagem `prepareok` correspondente a esse pedido para o primário.

```
...
replica(Central) ->
...
    Message = message:recv([{prepare},{commit}], 15),
    {Label,{}} = Message,
    if
        Label == timeout ->
            if
                Status == normal -> viewchange(Central);
                true -> ok
            end;

        Label == prepare ->
            replica(preparePhase, Central, Message);

        Label == commit ->
            replica(commitPhase, Central, Message)
    end,
...
```

Quando o processo recebe uma mensagem `commit` é chamada a função `replica` que recebe o argumento `commitPhase`. Nessa função tenta-se fazer *commit* da operação e depois volta-se ao início para receber mais mensagens. Caso ocorra um *timeout* é chamada a função `viewchange` do módulo `viewchang_xse` que dá inicio à troca de primário.

Esta função `recv` corresponde à combinação dos estados *PreparePhase* e *Commit-*

Phase do protocolo local *VR_rs*. Na implementação decidimos juntar a receção desta duas mensagens no mesmo `recv`, pois a réplica pode receber qualquer uma das mensagens a qualquer altura. No entanto continua a ser possível verificar que estas opções seguem o mesmo caminho que no protocolo local. Na opção *prepare* do estado *PreparePhase* a comunicação avança para o estado *PrepareOkPhase* e o mesmo acontece na implementação. Na opção *commit* do estado *CommitPhase* a instância do protocolo termina e na implementação volta-se ao ínicio onde se espera por mais pedidos. A opção *timeout* em ambos os estado leva ao estado *ViewChangePhase* do protocolo local *ViewChange_xs*, tal como acontece na implementação.

Por último, no estado *RetryPhase* do protocolo local *VR_rs* é recebida uma mensagem *request* do cliente. Segundo a descrição do protocolo VR as réplicas ignoram esta mensagem, pois esta só tem significado para o primário. Portanto na implementação esta mensagens também é ignorada, caso seja recebida o processo irá continuar no *recv* inicial à espera de mensagens *prepare* ou *commit*. No protocolo local quando a mensagem *request* é recebida a comunicação avança para o estado *PreparePhase*, mas o estado *CommitPhase* pode ser chamado a qualquer altura.

5.3 Discussão

Neste capítulo apresentámos um API de comunicação da linguagem *Erlang*, que permite programar sistemas distribuídos tolerantes a falhas. Também exemplificámos como se usa esta API e comparámos a implementação dos protocolos com os protocolos locais.

Esta comparação permite perceber que o programa e os protocolos locais apresentam uma correspondência em termos de comunicação, o que nos leva a acreditar que a nossa linguagem de protocolos é suficientemente poderosa para conseguir especificar a comunicação dos sistemas distribuídos tolerantes a falhas.

A API que criámos é um bom modelo para seguir quando se pretende implementar um sistema distribuído tolerante a falhas de forma rápida e fácil. No entanto não conseguimos fazer a verificação dos programas em tempo de compilação, nem era nosso objetivo, por isso os programas apresentados foram todos testados, recorrendo a métodos tradicionais.

Capítulo 6

Conclusão

Este relatório, enquadrado num Projeto em Engenharia Informática (PEI), contribui para um tópico muito importante que é a implementação de sistemas distribuídos tolerantes a falhas. Hoje em dia a maioria dos sistemas envolvem trocas de mensagens. Deste modo, existe sempre a possibilidade de haver alguma falha que afete o curso normal da comunicação. Por isso encontrar uma forma de implementar estes sistemas de forma rápida, fácil e correta é muito vantajoso para os projetos de desenvolvimento de software para sistemas distribuídos.

Existem várias abordagens para detetar erros de comunicação nestes programas, sendo a verificação em tempo de execução a mais utilizada, mas as técnicas existentes ainda apresentam alguns problemas, como a perda de desempenho do sistema. Já existem várias técnicas para detetar esses erros em tempo de compilação o que é vantajoso, pois consegue-se fazer uma deteção mais cedo e mais rápida, mas as técnicas existentes ainda não se encontram preparadas para verificar sistemas que toleram falhas.

Assim o nosso PEI contribui para esse campo que ainda se encontra por explorar, oferecendo uma linguagem poderosa que permite especificar os sistemas distribuídos tolerantes a falhas e um modelo geral para implementar a sua comunicação, que em conjunto poderão ser a chave para a verificação deste sistemas em tempo de compilação.

As três tarefas principais deste PEI foram concluídas com sucesso. Foram elas:

- Criação de um linguagem capaz de especificar os apetos da comunicação dos sistemas distribuídos tolerantes a falhas.

- Definição de regras de tradução que permitem fazer a projeção dos protocolos globais, que representam toda a comunicação do sistema de um ponto de vista global, para protocolo locais, que oferecem uma vista individual de cada processo envolvido no sistema.

- Investigação de linhas gerais para verificar que os programas estão de acordo com os protocolos locais. Para tal criámos uma API que oferece um modelo para implementar a comunicação de sistemas distribuídos tolerantes a falhas e que permite

estabelecer uma correspondência entre os protocolos locais e o programa.

Um trabalho futuro que propomos é usar os protocolos locais e os programas que criámos para verificar parcialmente sistemas distribuídos tolerantes a falhas em tempo de compilação. Para tal seria necessário construir uma ferramenta para fazer a tradução automática de protocolo global para protocolo local. Essa ferramenta teria que verificar se o protocolo global está bem construído, isto é, não existem erros de sintaxe, todas as variáveis e estados estão declarados, não existem ciclos, entre outras propriedades. Assim esta ferramenta devolveria protocolos locais bem construídos, obtidos através da regras de tradução usadas de encontro ao protocolo global bem formado. Tendo em conta que os protocolo locais estariam bem formados, poderiam ser asseguradas várias propriedades, como a ausência de pontos de bloqueio.

Depois teria que se criar um verificador para verificar em tempo de compilação que um dado programa segue um protocolo local. Uma das propriedades que se pretende verificar neste ponto é que uma determinada operação de envio ou receção que ocorre num determinado momento no programa leva a um ponto em que ocorre outra operação, e estas operações têm que corresponder às apresentadas no protocolo local. Por exemplo, quando num protocolo local existe um envio de uma mensagem a que leva a um estado onde é recebida uma mensagem b, então no programa após ser enviada a mensagem a haverá um receção de uma mensagem b.

Com a possibilidade de fazer a verificação deste programas em tempo de compilação, a programação de sistemas distribuídos tolerantes a falhas seria facilitada, pois pelo menos em termos de comunicação já seria possível saber antecipadamente se as operações têm uma correspondência em cada lado da comunicação e se as mensagens trocadas são as esperadas ou não.

Appendices

Apêndice A

API de comunicação para Erlang

```
-module(message).
-export([send/2, multisend/2, recv/2, multireceive/2]).

send(Message, Pid) ->
  Pid ! Message.

multisend(_, []) ->
  messageSent;

multisend(Message, [Pid|Pids]) ->
  Pid ! Message,
  multisend(Message, Pids).

recv([{Label}], Timeout) when Timeout == 0 ->
  receive
    {Label,Params} -> {Label,Params}
  end;

recv([{Label1},{Label2}], Timeout) when Timeout == 0 ->
  receive
    {Label1,Params1} -> {Label1,Params1};
    {Label2,Params2} -> {Label2,Params2}
  end;

recv([{Label1},{Label2},{Label3}], Timeout) when Timeout == 0 ->
  receive
    {Label1,Params1} -> {Label1,Params1};
    {Label2,Params2} -> {Label2,Params2};
    {Label3,Params3} -> {Label3,Params3}
  end;

recv([{Label}], Timeout) when Timeout > 0 ->
  receive
    {Label,Params} -> {Label,Params}
  after Timeout*1000 ->
    {timeout,{}}
  end;

recv([{Label1},{Label2}], Timeout) when Timeout > 0 ->
  receive
```

```
    {Label1,Params1} -> {Label1,Params1};
    {Label2,Params2} -> {Label2,Params2}
  after Timeout*1000 ->
    {timeout,{}}
  end;

recv([{Label1},{Label2},{Label3}], Timeout) when Timeout > 0 ->
  receive
    {Label1,Params1} -> {Label1,Params1};
    {Label2,Params2} -> {Label2,Params2};
    {Label3,Params3} -> {Label3,Params3}
  after Timeout*1000 ->
    {timeout,{}}
  end.

multireceive (LOptions, Timeout) ->
  if
    Timeout > 0 -> Ref = timer:send_after(1000*Timeout, timeout);
    Timeout == 0 -> Ref = false
  end,
  Opt1 = lists:nth(1,LOptions),
  if
    tuple_size(Opt1) == 3 ->
      multireceive(LOptions, [], Timeout, Ref);
    tuple_size(Opt1) == 2 ->
      LCounter = counter_list(LOptions, []),
      multireceive(LOptions, LCounter, [], Timeout, Ref)
  end.

multireceive([{Label,NMsg}], LCounter, LMessages, Timeout, Ref) when
    Timeout == 0 ->
  receive
    {Label,Params} -> is_necessary([{Label,NMsg}], LCounter, LMessages,
        {Label,Params}, Timeout, Ref)
  end;

multireceive([{Label1,NMsg1},{Label2,NMsg2}], LCounter, LMessages,
    Timeout, Ref) when Timeout == 0 ->
  LOptions = [{Label1,NMsg1},{Label2,NMsg2}],
  receive
    {Label1,Params1} -> is_necessary(LOptions, LCounter, LMessages, {
        Label1,Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LCounter, LMessages, {
        Label2,Params2}, Timeout, Ref)
  end;

multireceive([{Label1,NMsg1},{Label2,NMsg2},{Label3,NMsg3}], LCounter,
    LMessages, Timeout, Ref) when Timeout == 0 ->
  LOptions = [{Label1,NMsg1},{Label2,NMsg2},{Label3,NMsg3}],
  receive
    {Label1,Params1} -> is_necessary(LOptions, LCounter, LMessages, {
        Label1,Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LCounter, LMessages, {
        Label2,Params2}, Timeout, Ref);
    {Label3,Params3} -> is_necessary(LOptions, LCounter, LMessages, {
        Label3,Params3}, Timeout, Ref)
```

```
  end;

multireceive([{Label,NMsg}], LCounter, LMessages, Timeout, Ref) when
    Timeout > 0 ->
  receive
    {Label,Params} -> is_necessary([{Label,NMsg}], LCounter, LMessages,
        {Label,Params}, Timeout, Ref);
    timeout -> {timeout,{}}
  end;

multireceive([{Label1,NMsg1},{Label2,NMsg2}], LCounter, LMessages,
    Timeout, Ref) when Timeout > 0 ->
  LOptions = [{Label1,NMsg1},{Label2,NMsg2}],
  receive
    {Label1,Params1} -> is_necessary(LOptions, LCounter, LMessages, {
       Label1,Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LCounter, LMessages, {
       Label2,Params2}, Timeout, Ref);
    timeout -> {timeout,{}}
  end;

multireceive([{Label1,NMsg1},{Label2,NMsg2},{Label3,NMsg3}], LCounter,
    LMessages, Timeout, Ref) when Timeout > 0 ->
  LOptions = [{Label1,NMsg1},{Label2,NMsg2},{Label3,NMsg3}],
  receive
    {Label1,Params1} -> is_necessary(LOptions, LCounter, LMessages, {
       Label1,Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LCounter, LMessages, {
       Label2,Params2}, Timeout, Ref);
    {Label3,Params3} -> is_necessary(LOptions, LCounter, LMessages, {
       Label3,Params3}, Timeout, Ref);
    timeout -> {timeout,{}}
  end.

multireceive([{Label,NMsg,Function}], LMessages, Timeout, Ref) when
    Timeout == 0 ->
  LOptions = [{Label, NMsg, Function}],
  receive
    {Label,Params} -> is_necessary(LOptions, LMessages, {Label,Params},
        Timeout, Ref)
  end;

multireceive([{Label1,NMsg1,Func1},{Label2,NMsg2,Func2}], LMessages,
    Timeout, Ref) when Timeout == 0 ->
  LOptions = [{Label1,NMsg1,Func1},{Label2,NMsg2,Func2}],
  receive
    {Label1,Params1} -> is_necessary(LOptions, LMessages, {Label1,
       Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LMessages, {Label2,
       Params2}, Timeout, Ref)
  end;

multireceive([{Label1,NMsg1,Func1},{Label2,NMsg2,Func2},{Label3,NMsg3,
    Func3}], LMessages, Timeout, Ref) when Timeout == 0 ->
  LOptions = [{Label1,NMsg1,Func1},{Label2,NMsg2,Func2},{Label3,NMsg3,
     Func3}],
```

```
  receive
    {Label1,Params1} -> is_necessary(LOptions, LMessages, {Label1,
       Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LMessages, {Label2,
       Params2}, Timeout, Ref);
    {Label3,Params3} -> is_necessary(LOptions, LMessages, {Label3,
       Params3}, Timeout, Ref)
  end;

multireceive([{Label,NMsg,Function}], LMessages, Timeout, Ref) when
    Timeout > 0 ->
  LOptions = [{Label, NMsg, Function}],
  receive
    {Label,Params} -> is_necessary(LOptions, LMessages, {Label,Params},
        Timeout, Ref);
    timeout -> {timeout,{}}
  end;

multireceive([{Label1,NMsg1,Func1},{Label2,NMsg2,Func2}], LMessages,
    Timeout, Ref) when Timeout > 0 ->
  LOptions = [{Label1,NMsg1,Func1},{Label2,NMsg2,Func2}],
  receive
    {Label1,Params1} -> is_necessary(LOptions, LMessages, {Label1,
       Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LMessages, {Label2,
       Params2}, Timeout, Ref);
    timeout -> {timeout,{}}
  end;

multireceive([{Label1,NMsg1,Func1},{Label2,NMsg2,Func2},{Label3,NMsg3,
    Func3}], LMessages, Timeout, Ref) when Timeout > 0 ->
  LOptions = [{Label1,NMsg1,Func1},{Label2,NMsg2,Func2},{Label3,NMsg3,
     Func3}],
  receive
    {Label1,Params1} -> is_necessary(LOptions, LMessages, {Label1,
       Params1}, Timeout, Ref);
    {Label2,Params2} -> is_necessary(LOptions, LMessages, {Label2,
       Params2}, Timeout, Ref);
    {Label3,Params3} -> is_necessary(LOptions, LMessages, {Label3,
       Params3}, Timeout, Ref);
    timeout -> {timeout,{}}
  end.

is_necessary(LOptions, LMessages, Message, Timeout, Ref) ->
  {Label,_} = Message,
  NewLMessages = lists:append(LMessages, [Message]),
  Option = lists:keyfind(Label,1,LOptions),
  {_, Necessary, {Module,Name}} = Option,
  List = Module:Name(NewLMessages),
  if
    length(List) >= Necessary ->
      cancel_timer(Ref),
      List;
    length(List) < Necessary ->
      multireceive(LOptions, NewLMessages, Timeout, Ref)
  end.
```

```
is_necessary(LOptions, LCounter, LMessages, Message, Timeout, Ref) ->
  {Label,_} = Message,
  {_, Necessary} = lists:keyfind(Label,1,LOptions),
  {_, Received} = lists:keyfind(Label,1,LCounter),
  NewLMessages = lists:append(LMessages, [Message]),
  if
    Necessary == Received+1 ->
      cancel_timer(Ref),
      clean_other_messages(NewLMessages, Label, []);
    Necessary /= Received+1 ->
      LCounter2 = lists:keyreplace(Label, 1, LCounter, {Label, Received
          +1}),
      multireceive(LOptions, LCounter2, NewLMessages, Timeout, Ref)
  end.

cancel_timer(Ref) when Ref /= false->
  {Res, Timer} = Ref,
  if
    Res == ok -> timer:cancel(Timer);
    true -> ok
  end;

cancel_timer(Ref) when Ref == false->
  ok.

counter_list([], LCounter) ->
  LCounter;

counter_list([{Label,_}|Options], LCounter) ->
  [{Label,0}|counter_list(Options, LCounter)].

clean_other_messages([], _, NewList) ->
  NewList;

clean_other_messages(LMessages, Label, NewList) ->
  [{LMsg,Params}|Messages] = LMessages,
  if
    LMsg /= Label ->
      clean_other_messages(Messages, Label, NewList);
    LMsg == Label ->
      NewList2 = lists:append(NewList, [{LMsg,Params}]),
      clean_other_messages(Messages, Label, NewList2)
  end.
```

Figura A.1: Funções da nossa API de comunicação para Erlang.

Apêndice B

Implementação do protocolo 2PC

```
-module(twophasecommit_c).
-export([coordinator/1]).

coordinator(LPids) ->
  coordinator(phaseOne, LPids).

coordinator(phaseOne, LPids) ->
  message:multisend({vote_request,{}}, LPids),
  coordinator(waitReady, LPids);

coordinator(waitReady, LPids) ->
  LMessages = message:multireceive([{vote_commit,length(LPids)},{
     vote_abort,1}], 10),
  if
    LMessages /= {timeout,{}} ->
      Vote = lists:keyfind(vote_abort,1,LMessages),
      if
        Vote == false ->
          coordinator(twoPhaseCommit, LPids);
        Vote /= false ->
          coordinator(twoPhaseAbort, LPids)
      end;

    LMessages == {timeout,{}} ->
      coordinator(twoPhaseAbort, LPids)
  end;

coordinator(twoPhaseCommit, LPids) ->
  message:multisend({global_commit,{}}, LPids);

coordinator(twoPhaseAbort, LPids) ->
  message:multisend({global_abort,{}}, LPids).
```

Figura B.1: Módulo twophasecommit_c.

```
-module(central).
-export([central/2]).

central(Coordinator, LParticipants) ->
  TwoPhase = spawn(twophasecommit_ps, participant, [self(), Coordinator
     , LParticipants]),
```

```
  Decision = spawn(decision_q, decision, [self()]),
  receive_cycle(Coordinator, LParticipants, TwoPhase, Decision,
     phaseOne).

receive_cycle(Coordinator, LParticipants, TwoPhase, Decision, State) ->
  receive
  {decision_request, {Pid}} ->  Decision ! {decision_request, {Pid}};

  {write, X} -> receive_cycle(Coordinator, LParticipants, TwoPhase,
     Decision, X);

  {read, Pid} -> Pid ! {state, State};

  Message ->  TwoPhase ! Message

  end,
  receive_cycle(Coordinator, LParticipants, TwoPhase, Decision, State).
```

Figura B.2: Módulo central.

```
-module(twophasecommit_ps).
-export([participant/3, get_state/1, set_state/2]).

participant(Central, Coordinator, LParticipants) ->
  State = get_state(Central),
  participant(State, Central, Coordinator, LParticipants).

participant(phaseOne, Central, Coordinator, LParticipants) ->
  {Label,_} = message:recv([{vote_request}], 10),
  if
    Label == vote_request ->
      set_state(Central, waitReadyVote);
    Label == timeout ->
      set_state(Central, waitReadyAbort)
  end,
  participant(Central, Coordinator, LParticipants);

participant(waitReadyVote, Central, Coordinator, LParticipants) ->
  Vote = chose([vote_commit,vote_abort]),
  message:send({Vote,{}}, Coordinator),
  set_state(Central, phaseTwo),
  participant(Central, Coordinator, LParticipants);

participant(waitReadyAbort, Central, Coordinator, LParticipants) ->
  message:send({vote_abort,{}}, Coordinator),
  set_state(Central, phaseTwo),
  participant(Central, Coordinator, LParticipants);

participant(phaseTwo, Central, Coordinator, LParticipants) ->
  {Label,_} = message:recv([{global_commit},{global_abort}], 10),
  if
    Label == global_commit ->
      set_state(Central, commit);
    Label == global_abort ->
      set_state(Central, abort);
    Label == timeout ->
```

```
      decision_p:decision(Central, Coordinator, LParticipants)
  end.

get_state(Pid) ->
  Pid ! {read, self()},
  receive
    {state, State} -> State
  end.

set_state(Pid, State) ->
  Pid ! {write, State}.
```

Figura B.3: Módulo twophasecommi_ps.

```
-module(decision_p).
-export([decision/3]).

decision(Central, _, []) ->
  twophasecommit_ps:set_state(Central, abort);

decision(Central, Coordinator, LParticipants) ->
  Pid = chose(LParticipants),
  message:send({decision_request,{self()}}, Pid),

  {Label2,_} = message:recv([{dec_global_commit},{dec_global_abort},{
      no_decision}], 10),
  if
    Label2 == dec_global_commit ->
      twophasecommit_ps:set_state(Central, commit);
    Label2 == dec_global_abort ->
      twophasecommit_ps:set_state(Central, abort);
    Label2 == no_decision ->
      decision(Central, Coordinator, LParticipants);
    Label2 == timeout ->
      decision(Central, Coordinator, LParticipants)
  end.

chose(LPart) ->
  {A1,A2,A3} = now(),
   random:seed(A1,A2,A3),
  Position = random:uniform(length(LPart)),
  lists:nth(Position, LPart).
```

Figura B.4: Módulo decision_p.

```
-module(decision_q).
-export([decision/1]).

decision(Central) ->
  {Label, {Pid}} = message:recv([{decision_request}], 0),

  State = twophasecommit_ps:get_state(Central),
  if
    State == commit ->
      message:send({dec_global_commit,{}}, Pid);
    State == phaseOne ->
```

```
      message:send({dec_global_abort,{}}, Pid),
      twophasecommit_ps:set_state(Central,abort);
    State == abort ->
      message:send({dec_global_abort,{}}, Pid);
    State == phaseTwo ->
      message:send({no_decision,{}}, Pid)
  end.
```

Figura B.5: Módulo decision_q.

Apêndice C

Implementação do protocolo VR

```
-module(vr_c).
-export([start/2, get_primary/2]).

start(Replicas, ClientID) ->
  {ViewNumber,RequestNumber} = read_file(ClientID),
  client(Replicas, ClientID, ViewNumber, RequestNumber).

client(Replicas, ClientID, ViewNumber, RequestNumber) ->
  {ok, [OperationId]} = io:fread("Inserir id da operacao: ", "~d"),
  client(requestPhase, Replicas, ClientID, ViewNumber, RequestNumber,
     OperationId).

client(requestPhase, Replicas, ClientID, ViewNumber, RequestNumber,
    OperationId) ->
  Primary = get_primary(Replicas, ViewNumber),
  message:send({request, {OperationId, ClientID, RequestNumber}},
     Primary),
  client(replyPhase, Replicas, ClientID, ViewNumber, RequestNumber,
     OperationId);

client(replyPhase, Replicas, ClientID, ViewNumber, RequestNumber,
    OperationId) ->
  Message = message:recv([{reply}],10),
  {Label,Params} = Message,
  if
    Label == reply ->
      {ViewNumber2, RequestNumber2, Result} = Params,
      if
        RequestNumber2 == RequestNumber , Result > -1 ->
          io:format("Pedido numero ~p -> Resultado: ~p\n", [
              RequestNumber2,Result]),
          write_to_file(ViewNumber2, RequestNumber, ClientID),
          client(Replicas, ClientID, ViewNumber2, RequestNumber+1);
        RequestNumber2 > RequestNumber ->
          client(requestPhase, Replicas, ClientID, ViewNumber2,
              RequestNumber2+1, OperationId);
        true ->
          client(replyPhase, Replicas, ClientID, ViewNumber2,
              RequestNumber, OperationId)
      end;
```

```
    Label == timeout ->
        client(retryPhase, Replicas, ClientID, ViewNumber,
           RequestNumber, OperationId)
  end;

client(retryPhase, Replicas, ClientID, ViewNumber, RequestNumber,
   OperationId) ->
  message:multisend({request, {OperationId, ClientID, RequestNumber}},
     Replicas),
  client(replyPhase, Replicas, ClientID, ViewNumber, RequestNumber,
     OperationId).

get_primary(Replicas, ViewNumber) ->
  Index = ViewNumber rem length(Replicas),
  lists:nth(Index+1, Replicas).

write_to_file(ViewNumber, RequestNumber, ClientID) ->
  Name = string:concat("conf", integer_to_list(ClientID)),
  {ok, File} = file:open(Name, [write]),
  io:format(File, "~p\n~p",[ViewNumber, RequestNumber]),
  file:close(File).

read_file(ClientID) ->
  Name = string:concat("conf", integer_to_list(ClientID)),
  {Label, X} = file:open(Name, [read]),

  if
    Label == error -> {0,1};
    Label == ok ->
      VN = io:get_line(X, ""),
      {ViewNumber,_} = string:to_integer(VN),
      RN = io:get_line(X, ""),
      {RequestNumber,_} = string:to_integer(RN),
      file:close(X),
      {ViewNumber,RequestNumber+1}
  end.
```

Figura C.1: Módulo vr_c.

```
-module(central).
-export([start/2, get_state/1, set_state/2]).

start(Clients, Replicas) ->
  State = state:createState(Clients, Replicas),

  %crias as threads
  ViewChange_rs = spawn(viewchange_rs, viewchange, [self()]),
  ViewChange_p = spawn(viewchange_p, viewchange, [self()]),
  Recovery_rs = spawn(recovery_rs, recovery, [self()]),
  Recovery_r = spawn(recovery_r, recovery, [self(), State]),
  StateTransfer_q = spawn(statetransfer_q, statetransfer, [self()]),

  %inicia
  Necessary = ((length(Replicas)-1)/2)+1,
  LMessages = message:multireceive([{recovery, Necessary}, {initiate,1}
     , {recoveryresponse, Necessary}], 15),
```

```
  if
    LMessages == {timeout,{}} ->
      io:fwrite("Nao conseguiu iniciar\n");

    true ->
      Message = lists:nth(1, LMessages),
      {Label,_} = Message,
      if
        Label == recovery ->
          OtherReplicas = state:get_replicas(State),
          message:multisend({initiate,{}}, OtherReplicas),
          NewState = state:set_in_list(status, normal, State);
        Label == initiate ->
          NewState = state:set_in_list(status, normal, State);
        Label == recoveryresponse ->
          send_to_recovery_r(LMessages, Recovery_r),
          receive
            {write, NewState} -> ok
          end
      end,
      exit(Recovery_r, ok),

      IsPrimary = state:is_primary(NewState),
      if
        IsPrimary == true ->
          HandleClient = spawn(vr_p_client, primary, [self()]),
          HandleReplicas = spawn(vr_p, primary, [self()]),
          central_p(HandleClient, HandleReplicas, ViewChange_rs,
              ViewChange_p, Recovery_rs, StateTransfer_q, NewState);

        IsPrimary /= true ->
          Replica = spawn(vr_rs, replica, [self()]),
          central_r(Replica, ViewChange_rs, ViewChange_p, Recovery_rs,
              StateTransfer_q, NewState)
      end
  end.

send_to_recovery_r([], _) -> ok;
send_to_recovery_r([Message|LMessages], Recovery_r) ->
  Recovery_r ! Message,
  send_to_recovery_r(LMessages, Recovery_r).

central_p(HandleClient, HandleReplicas, ViewChange_rs, ViewChange_p,
    Recovery, StateTransfer_q, State) ->
  receive
    Message -> ok
  end,

  {Label,_} = Message,
  Status = state:get_from_list(status, State),
  IsRecoveryMessage = is_recovery_message(Label),
  IsViewChangeMessage = is_viewchange_message(Label),
  if
    Status == recovering , IsRecoveryMessage == false ->
      central_p(HandleClient, HandleReplicas, ViewChange_rs,
```

```
          ViewChange_p, Recovery, StateTransfer_q, State);
    Status == viewchange , IsViewChangeMessage == false ->
      central_p(HandleClient, HandleReplicas, ViewChange_rs,
          ViewChange_p, Recovery, StateTransfer_q, State);
    true -> ok
  end,

  case Label of
    request-> HandleClient ! Message;

    prepareok -> HandleReplicas ! Message;

    startviewchange -> ViewChange_rs ! Message;

    doviewchange -> ViewChange_p ! Message;

    startview -> ViewChange_rs ! Message;

    recovery -> Recovery ! Message;

    getstate -> StateTransfer_q ! Message;

    change ->
      exit(HandleClient, ok),
      exit(HandleReplicas, ok),
      Replica = spawn(vr_rs, replica, [self()]),
      central_r(Replica, ViewChange_rs, ViewChange_p, Recovery,
          StateTransfer_q,  State);

    write ->
      {write, X} = Message,
      central_p(HandleClient, HandleReplicas, ViewChange_rs,
          ViewChange_p, Recovery, StateTransfer_q, X);

    read ->
      {read, Pid} = Message,
      Pid ! {state, State};

    _ELSE -> ok

  end,
  central_p(HandleClient, HandleReplicas, ViewChange_rs, ViewChange_p,
      Recovery, StateTransfer_q, State).

central_r(Replica, ViewChange_rs, ViewChange_p, Recovery,
    StateTransfer_q, State) ->
  receive
    Message -> ok
  end,

  {Label,_} = Message,
  Status = state:get_from_list(status, State),
  IsRecoveryMessage = is_recovery_message(Label),
  IsViewChangeMessage = is_viewchange_message(Label),
  if
    Status == recovering , IsRecoveryMessage == false ->
```

```
      central_r(Replica, ViewChange_rs, ViewChange_p, Recovery,
         StateTransfer_q, State);
    Status == viewchange , IsViewChangeMessage == false ->
      central_r(Replica, ViewChange_rs, ViewChange_p, Recovery,
         StateTransfer_q, State);
    true -> ok
  end,

  case Label of

    prepare -> Replica ! Message;

    commit -> Replica ! Message;

    startviewchange -> ViewChange_rs ! Message;

    doviewchange -> ViewChange_p ! Message;

    startview -> ViewChange_rs ! Message;

    recovery -> Recovery ! Message;

    recoveryresponse -> Replica ! Message;

    getstate -> StateTransfer_q ! Message;

    newstate -> Replica ! Message;

    change ->
      exit(Replica, ok),
      HandleClient = spawn(vr_p_client, primary, [self()]),
      HandleReplicas = spawn(vr_p, primary, [self()]),
      central_p(HandleClient, HandleReplicas, ViewChange_rs,
         ViewChange_p, Recovery, StateTransfer_q, State);

    write ->
      {write, X} = Message,
      central_r(Replica, ViewChange_rs, ViewChange_p, Recovery,
         StateTransfer_q, X);

    read ->
      {read, Pid} = Message,
      Pid ! {state, State};

    _ELSE -> ok

  end,
  central_r(Replica, ViewChange_rs, ViewChange_p, Recovery,
     StateTransfer_q, State).

get_state(Central) ->
  Central ! {read,self()},
  receive {state, State} -> State end.

set_state(_, false) -> ok;
set_state(Central, State) ->
```

```
  Central ! {write, State}.

is_recovery_message(Label) ->
  if
    Label == recoveryresponse -> true;
    Label == read -> true;
    Label == write -> true;
    true -> false
  end.

is_viewchange_message(Label) ->
  if
    Label == startviewchange -> true;
    Label == startview -> true;
    Label == doviewchange -> true;
    Label == write -> true;
    Label == read -> true;
    Label == change -> true;
    true -> false
  end.
```

Figura C.2: Módulo central.

```
-module(vr_p_client).
-export([primary/1]).

primary(Central) ->
  primary(requestPhase, Central),
  primary(Central).

primary(requestPhase, Central) ->
  {Label,Params} = message:recv([{request}], 10),
  if
    Label == timeout ->
      primary(commitPhase, Central);
    Label == request -> ok
  end,

  {_, ClientID, RequestNumber} = Params,
  State = central:get_state(Central),
  IsDuplicate = state:is_duplicate(RequestNumber, ClientID, State),
  if
    IsDuplicate == true ->
      LastRequestNumber = state:get_last_requestnumber(ClientID, State)
          ,
      LastResult = state:get_last_result(ClientID, State),
      ClientPid = state:get_client_pid(ClientID, State),
      ViewNumber = state:get_from_list(viewnumber, State),

      if
        LastRequestNumber == RequestNumber , LastResult /= -1 ->
          message:send({reply,{ViewNumber, RequestNumber, LastResult}},
              ClientPid);
        true ->
          message:send({reply,{ViewNumber, LastRequestNumber,-1}},
             ClientPid)
```

```
      end;
    IsDuplicate == false ->
        OpNumber = state:get_from_list(opnumber, State),
        State1 = state:inc_value(opnumber, State),
        State2 = state:add_to_log({OpNumber+1, {Label,Params}}, State1)
            ,
        NewState = state:set_client_requestnumber(ClientID,
            RequestNumber, State2),
        central:set_state(Central, NewState),
        primary(preparePhase, NewState, {Label,Params})
  end;

primary(commitPhase, Central) ->
  State = central:get_state(Central),
  ViewNumber = state:get_from_list(viewnumber, State),
  CommitNumber = state:get_from_list(commitnumber, State),
  message:multisend({commit, {ViewNumber, CommitNumber}}, state:
      get_replicas(State)),
  primary(Central).

primary(preparePhase, State, Request) ->
  ViewNumber = state:get_from_list(viewnumber, State),
  OpNumber = state:get_from_list(opnumber, State),
  CommitNumber = state:get_from_list(commitnumber, State),
  Replicas = state:get_replicas(State),
  message:multisend({prepare, {ViewNumber, Request, OpNumber,
      CommitNumber}}, Replicas).
```

Figura C.3: Módulo vr_p_client.

```
-module(vr_p).
-export([primary/1, filter_prepareok/1, filter/2]).

primary(Central) ->
  primary(prepareOkPhase, Central),
  primary(Central).

primary(prepareOkPhase, Central) ->
  State1 = central:get_state(Central),
  Necessary = state:necessary(State1),
  LMessages = message:multireceive([{prepareok,Necessary,{vr_p,
      filter_prepareok}}], 0),
  State = central:get_state(Central),
  {prepareok, {_, OpNumber, _}} = lists:nth(1,LMessages),

  CommitNumber = state:get_from_list(commitnumber, State),
  if
    OpNumber =< CommitNumber -> done;
    OpNumber > CommitNumber ->
      Result = state:commit(Central, OpNumber, State),
      State2 = central:get_state(Central),
      if
        Result /= false -> primary(replyPhase, State2, OpNumber, Result
            );
        true -> ok
      end
```

```
  end.

primary(replyPhase, State, OpNumber, Result) ->
  {_,{_,ClientID,RequestNumber}} = state:get_request(OpNumber, State),
  ClientPid = state:get_client_pid(ClientID, State),
  ViewNumber = state:get_from_list(viewnumber, State),
  message:send({reply,{ViewNumber,RequestNumber,Result}}, ClientPid).

filter_prepareok(LMessages) ->
  filter(LMessages,2).

filter(LMessages, Position) ->
  {Label,Params} = lists:last(LMessages),
  Element = element(Position,Params),
  lists:filter(fun(X) -> element(1,X) == Label end,
  lists:filter(fun(X) -> element(Position,element(2,X)) == Element end,
      LMessages)).
```

Figura C.4: Módulo vr_p.

```
-module(vr_rs).
-export([replica/1]).

replica(Central) ->
  State = central:get_state(Central),
  Status = state:get_from_list(status, State),

  Message = message:recv([{prepare},{commit}], 15),
  {Label,_} = Message,
  if
    Label == timeout ->
      if
        Status == normal -> viewchange_xs:viewchange(Central);
        true -> ok
      end;

    Label == prepare ->
      replica(preparePhase, Central, Message);

    Label == commit ->
      replica(commitPhase, Central, Message);

    true -> label_errada
  end,

  replica(Central).

replica(preparePhase, Central, Message) ->
  State = central:get_state(Central),
  {prepare, {_, Request, OpNumber, CommitNumber}} = Message,

  LastOpNumber = state:get_from_list(opnumber, State),
  if
    OpNumber-1 == LastOpNumber ->
      State2 = state:inc_value(opnumber, State),
      State3 = state:add_to_log({OpNumber, Request}, State2),
```

```
      {_,{_,ClientID,RequestNumber}} = Request,
      State4 = state:set_client_requestnumber(ClientID, RequestNumber,
         State3),
      central:set_state(Central, State4),
      replica(prepareOkPhase, Central, Message),
      LastCommitNumber = state:get_from_list(commitnumber,State4),
      if
        CommitNumber > LastCommitNumber ->
          state:commit(Central, CommitNumber, State4);
        CommitNumber =< LastCommitNumber -> ok
      end;

    OpNumber-1 > LastOpNumber ->
      Replicas = state:get_replicas(State),
      replica(stateTransfer, Central, Replicas);

    true -> ok
  end;

replica(prepareOkPhase, Central, Message) ->
  State = central:get_state(Central),
  ViewNumber = state:get_from_list(viewnumber, State),
  {_, {_, _, OpNumber, _}} = Message,
  ReplicaID = state:get_replica_ID(State),
  Primary = state:get_primary(State),
  message:send({prepareok, {ViewNumber, OpNumber, ReplicaID}}, Primary)
     ;

replica(commitPhase, Central, Message) ->
  {commit, {_, CommitNumber}} = Message,
  State = central:get_state(Central),
  MyCommitNumber = state:get_from_list(commitnumber, State),
  if
    MyCommitNumber < CommitNumber ->
      state:commit(Central, CommitNumber, State);
    true -> ok
  end.
```

Figura C.5: Módulo vr_rs.

```
-module(viewchange_xs).
-export([viewchange/1]).

viewchange(Central) ->
  State = central:get_state(Central),
  Status = state:get_from_list(status, State),
  if
    Status == normal ->
      LastestViewNumber = state:get_from_list(viewnumber, State),
      State2 = state:set_in_list(lastestviewnumber, LastestViewNumber,
         State),
      State3 = state:set_in_list(status, viewchange, State2),
      NewState = state:inc_value(viewnumber, State3);
    Status == viewchange ->
      NewState = state:inc_value(viewnumber, State)
  end,
```

```
  central:set_state(Central, NewState),
  ReplicaID = state:get_replica_ID(NewState),
  ViewNumber = state:get_from_list(viewnumber, NewState),
  Replicas = state:get_replicas(NewState),
  message:multisend({startviewchange, {ViewNumber, ReplicaID}},
     Replicas).
```

Figura C.6: Módulo viewchange_xs.

```
-module(viewchange_rps).
-export([viewchange/1, filter_viewchange/1]).

viewchange(Central) ->
  viewchange(viewChange, Central),
  viewchange(Central).

viewchange(viewChange, Central) ->
  State = central:get_state(Central),
  Necessary = state:necessary(State),
  LMessages = message:multireceive([{startviewchange,Necessary,{
     viewchange_rs,filter_viewchange}}], 0),
  State2 = central:get_state(Central),
  Status = state:get_from_list(status, State2),
  {startviewchange, {ViewNumber, _}} = lists:nth(1,LMessages),
  MyViewNumber = state:get_from_list(viewnumber, State2),

  if
    Status == normal ->
      viewchange_xs:viewchange(Central);
    Status == viewchange -> ok
  end,

  if
    ViewNumber == MyViewNumber ->
      viewchange(doViewChange, Central);
    ViewNumber > MyViewNumber ->
      viewchange_xs:viewchange(Central)
  end;

viewchange(doViewChange, Central) ->
  State = central:get_state(Central),
  ViewNumber = state:get_from_list(viewnumber, State),
  Log = state:get_from_list(log, State),
  LatestViewNumber = state:get_from_list(lastestviewnumber, State),
  OpNumber = state:get_from_list(opnumber, State),
  CommitNumber = state:get_from_list(commitnumber, State),
  ReplicaID = state:get_replica_ID(State),
  NewPrimary = state:get_primary(State),

  message:send({doviewchange, {ViewNumber, Log, LatestViewNumber,
     OpNumber, CommitNumber, ReplicaID}}, NewPrimary),

  IsPrimary = state:is_primary(State),
  if
    IsPrimary == true -> viewchange(Central);
```

```
    IsPrimary == false -> viewchange(startView, Central)
  end;

viewchange(startView, Central) ->
  {Label,Params} = message:recv([{startview}], 10),
  if
    Label == timeout -> viewchange_xs:viewchange(Central);
    Label /= timeout -> viewchange(startView, Central, {Label,Params})
  end.

viewchange(startView, Central, Message) ->
  State = central:get_state(Central),
  LatestViewNumber = state:get_from_list(lastestviewnumber, State),

  {startview, {ViewNumber, Log, OpNumber, CommitNumber}} = Message,
  if
    Log /= [] ->
      {LastOpNumber, _} = lists:max(Log),
      State2 = state:set_in_list(opnumber, LastOpNumber, State);
    Log == [] ->
      State2 = state:set_in_list(opnumber, OpNumber, State)
  end,
  State3 = state:set_in_list(log, Log, State2),
  State4 = state:set_in_list(viewnumber, ViewNumber, State3),
  State5 = state:set_in_list(status, normal, State4),
  central:set_state(Central,State5),

  prepare_old_operations(OpNumber, CommitNumber, Log, ViewNumber,
     State5),
  NewState = state:commit_old_operations(Central, CommitNumber, State5)
     ,

  Replicas = state:get_from_list(replicas, NewState),
  Pid = lists:nth(LatestViewNumber+1, Replicas),
  if
    node() == Pid -> Central ! {change};
    true -> ok
  end,

  central:set_state(Central,NewState),
  viewchange(Central).

prepare_old_operations(OpNumber, CommitNumber, Log, ViewNumber, State)
   ->
  ReplicaID = state:get_replica_ID(State),
  Primary = state:get_primary(State),
  if
    OpNumber > CommitNumber ->
      message:send({prepareok,{ViewNumber,CommitNumber+1,ReplicaID}},
         Primary),
      prepare_old_operations(OpNumber, CommitNumber+1, Log, ViewNumber,
          State);
    OpNumber == CommitNumber ->
      ok
  end.
```

```
filter_viewchange(LMessages) ->
  vr_p:filter(LMessages,1).
```

Figura C.7: Módulo viewchange_rps.

```
-module(viewchange_p).
-export([viewchange/1, filter_doviewchange/1]).

viewchange(Central) ->
  viewchange(doViewChange, Central).

viewchange(doViewChange, Central) ->
  State = central:get_state(Central),
  Necessary = state:necessary(State),
  LMessages = message:multireceive([{doviewchange,Necessary+1,{
     viewchange_p,filter_doviewchange}}], 0),
  io:fwrite("DoViewChange no p\n"),

  State1 = central:get_state(Central),
  Status = state:get_from_list(status, State1),
  {_,{ViewNumber,_,_,_,_,_}} = lists:nth(1,LMessages),

  if
    Status == normal ->
      viewchange(doViewChange, Central);
    Status == viewchange -> ok
  end,
  State2 = state:set_in_list(viewnumber, ViewNumber, State1),
  Log = get_log_of_bigger_viewnumber(LMessages),
  State3 = state:set_in_list(log, Log, State2),
  State4 = state:set_in_list(status, normal, State3),
  if
    Log == [] -> OpNumber = get_bigger_opnumber(LMessages);
    Log /= [] -> {OpNumber, _} = lists:max(Log)
  end,
  State5 = state:set_in_list(opnumber, OpNumber, State4),
  CommitNumber = get_bigger_commitnumber(LMessages),
  State6 = state:set_in_list(commitnumber, CommitNumber, State5),
  NewState = state:commit_old_operations(Central, CommitNumber, State6)
     ,
  central:set_state(Central, NewState),
  viewchange(startView, Central, NewState).

viewchange(startView, Central, State) ->
  ViewNumber = state:get_from_list(viewnumber, State),
  Log = state:get_from_list(log, State),
  OpNumber = state:get_from_list(opnumber, State),
  CommitNumber = state:get_from_list(commitnumber, State),
  Replicas = state:get_replicas(State),
  message:multisend({startview, {ViewNumber, Log, OpNumber,
     CommitNumber}}, Replicas),

  Central ! {change,{}},
  central:set_state(Central, State),
  viewchange(Central).
```

```
filter_doviewchange(LMessages) ->
  vr_p:filter(LMessages,1).

get_log_of_bigger_viewnumber(LMessages) ->
  LViewNumber = get_list_of_counters(LMessages, 3),
  MaxViewNumber = lists:max(LViewNumber),
  NewLMessages = lists:filter(fun(X) -> element(3,element(2,X)) ==
     MaxViewNumber end, LMessages),

  if
    length(NewLMessages) > 1 ->
      MaxOpNumber = get_bigger_opnumber(NewLMessages),
      List = lists:filter(fun(X) -> element(4,element(2,X)) ==
         MaxOpNumber end, NewLMessages),
      {_,{_,Log,_,_,_,_}} = lists:nth(1, List);
    length(NewLMessages) == 1 ->
      {_,{_,Log,_,_,_,_}} = lists:nth(1, NewLMessages)
  end,
  Log.

get_bigger_opnumber(LMessages) ->
  Lists = get_list_of_counters(LMessages, 4),
  lists:max(Lists).

get_bigger_commitnumber(LMessages) ->
  List = get_list_of_counters(LMessages, 5),
  lists:max(List).

get_list_of_counters([], _) -> [];
get_list_of_counters([{_,Params}|LMessages], Position) ->
  [element(Position, Params)|get_list_of_counters(LMessages, Position)]
     .
```

Figura C.8: Módulo viewchange_p.

```
-module(recovery_r).
-export([recovery/2, filter_recoveryresponse/1]).

recovery(Central, State) ->
  recovery(recovery, Central, State),
  recovery(Central, State).

recovery(recovery, Central, State) ->
  Replicas = state:get_replicas(State),
  ReplicaID = state:get_replica_ID(State),
  Nonce = generate_nonce(ReplicaID),
  message:multisend({recovery, {ReplicaID, Nonce}}, Replicas),
  recovery(recoveryResponse, Central, State, Nonce).

recovery(recoveryResponse, Central, State, Nonce) ->
  Necessary = state:necessary(State)+1,
  LMessages = message:multireceive([{recoveryresponse,Necessary,{
     recovery_r,filter_recoveryresponse}}], 10),

  if
    LMessages == {timeout,{}} ->
```

```
      recovery(Central, State);
    true ->
      {_,Params} = lists:nth(1,LMessages),
      {ViewNumber,NonceReceived,_,_,_,_} = Params,
      if
        NonceReceived == Nonce ->
          {_,{ViewNumber2,_,Log,OpNumber,CommitNumber,_}} =
              get_message_from_primary(LMessages, ViewNumber),
          State2 = state:set_in_list(viewnumber, ViewNumber2, State),
          State3 = state:set_in_list(log, Log, State2),
          State4 = state:set_in_list(opnumber, OpNumber, State3),
          State5 = state:set_in_list(commitnumber, CommitNumber, State4
              ),
          State6 = state:set_in_list(status, normal, State5),
          NewState = state:update_clienttable(Log, State6),
          central:set_state(Central, NewState),
          recovery(Central, NewState);
        true ->
          recovery(recoveryResponse, Central, State, Nonce)
      end
  end.

get_message_from_primary([], _) ->
  false;
get_message_from_primary([{Label,Params}|Messages], ViewNumber) ->
  ReplicaID = element(6,Params),
  if
    ReplicaID == ViewNumber+1 ->
       {Label,Params};
    true ->
      get_message_from_primary(Messages, ViewNumber)
  end.

filter_recoveryresponse(LMessages) ->
  NewLMessages = vr_p:filter(LMessages, 2),
  {_,{ViewNumber,_,_,_,_,_}} = lists:nth(1,NewLMessages),
  Msg = get_message_from_primary(LMessages, ViewNumber),
  if
    Msg == false -> [];
    true -> NewLMessages
  end.

generate_nonce(ReplicaID) ->
  calendar:time_to_seconds(time()) + ReplicaID.
```

Figura C.9: Módulo recovery_r.

```
-module(recovery_rs).
-export([recovery/1]).

recovery(Central) ->
  recovery(recovery, Central).

recovery(recovery, Central) ->
  {recovery, {ReplicaID, Nonce}} = message:recv([{recovery}], 0),
  recovery(recoveryResponse, Central, ReplicaID, Nonce),
```

```
  recovery(recovery, Central).

recovery(recoveryResponse, Central, ReplicaID, Nonce) ->
  State = central:get_state(Central),
  Status = state:get_from_list(status, State),
  if
    Status == normal ->
      IsPrimary = state:is_primary(State),
      ViewNumber = state:get_from_list(viewnumber, State),
      MyReplicaID = state:get_replica_ID(State),
      ReplicaPid = state:get_replica_Pid(State, ReplicaID),
      if
        IsPrimary == true ->
          Log = state:get_from_list(log,State),
          OpNumber = state:get_from_list(opnumber,State),
          CommitNumber = state:get_from_list(commitnumber,State),
          message:send({recoveryresponse, {ViewNumber, Nonce, Log,
          OpNumber, CommitNumber, MyReplicaID}}, ReplicaPid);
        IsPrimary == false ->
          message:send({recoveryresponse, {ViewNumber, Nonce, nil,
          nil, nil, MyReplicaID}}, ReplicaPid)
      end;

    true -> ok
  end.
```

Figura C.10: Módulo recovery_rs.

```
-module(statetransfer_r).
-export([statetransfer/1]).

statetransfer(Central) ->
  State = central:get_state(Central),
  ViewNumber = state:get_from_list(viewnumber, State),
  OpNumber = state:get_from_list(opnumber, State),
  ReplicaID = state:get_replica_ID(State),
  LReplicas = state:get_replicas(State),
  Replica = chose(LReplicas),

  message:send({getstate, {ViewNumber, OpNumber, ReplicaID}}, Replica),

  Message = message:recv([{newstate}], 15),
  if
    Message == timeout -> statetransfer(Central);
    true ->
      {newstate, {ViewNumber2, Log, OpNumber2, CommitNumber}} = Message
          ,
      State2 = state:set_in_list(viewnumber, ViewNumber2, State),
      State3 = state:set_in_list(log, Log, State2),
      State4 = state:set_in_list(opnumber, OpNumber2, State3),
      NewState = state:set_in_list(commitnumber, CommitNumber, State4),
      central:setstate(Central, NewState)
  end.

chose(List) ->
  {A1,A2,A3} = now(),
```

```
   random:seed(A1,A2,A3),
  Position = random:uniform(length(List)),
  lists:nth(Position, List).
```

Figura C.11: Módulo statetransfer_r.

```
-module(statetransfer_q).
-export([statetransfer/1]).

statetransfer(Central) ->
  Message = message:recv([{getstate}], 0),
  {_,{_,_,ReplicaID}} = Message,
  State = central:get_state(Central),
  ViewNumber = state:get_from_list(viewnumber, State),
  Log = state:get_from_list(log, State),
  OpNumber = state:get_from_list(opnumber, State),
  CommitNumber = state:get_from_list(commitnumber, State),
  ReplicaPid = state:get_replica_Pid(State, ReplicaID),
  message:send({newstate, {ViewNumber, Log, OpNumber, CommitNumber}},
     ReplicaPid).
```

Figura C.12: Módulo statetransfer_q.

```
-module(state).
-export([createState/2, is_duplicate/3, get_last_result/2,
    get_client_pid/2, get_from_list/2, set_in_list/3, inc_value/2,
    add_to_log/2, get_replicas/1, ready_commit/2, pos/2, commit/3,
    get_request/2, get_primary/1, get_replica_ID/1,
    get_last_requestnumber/2, set_client_requestnumber/3, is_primary/1,
     get_replica_Pid/2, get_client_info/2, necessary/1,
    update_clienttable/2, commit_old_operations/3]).

createState(Clients, Replicas) ->
  [{clients, Clients}, {replicas, Replicas}, {viewnumber, 0},
  {status, recovering}, {opnumber,0}, {commitnumber, 0},{
     lastestviewnumber, 0},
  {waitcommit, []}, {log,[]}, {clienttable, create_client_table(Clients
     )}].

create_client_table(Clients) ->
  create_client_table(Clients, 1).

create_client_table([], _) ->
  [];

create_client_table([_|Clients], Counter) ->
  [{Counter, 0, -1} | create_client_table(Clients, Counter+1)].

get_from_list(Label, List) ->
  {_, Content} = lists:keyfind(Label, 1, List),
  Content.

set_in_list(Label, NewContent, List) ->
  lists:keyreplace(Label, 1, List, {Label, NewContent}).

inc_value(Label, List) ->
```

```
  Value = get_from_list(Label, List),
  lists:keyreplace(Label, 1, List, {Label, Value+1}).

necessary(State) ->
  Replicas = get_from_list(replicas, State),
  (length(Replicas)-1)/2.

get_client_pid(ClientID, State) ->
  Clients = get_from_list(clients, State),
  lists:nth(ClientID, Clients).

get_replicas(State) ->
  Replicas = get_from_list(replicas, State),
  lists:keydelete(node(),2,Replicas).

get_replica_Pid(State, ReplicaID) ->
  Replicas = get_from_list(replicas, State),
  lists:nth(ReplicaID, Replicas).

get_primary(State) ->
  ViewNumber = get_from_list(viewnumber, State),
  Replicas = get_from_list(replicas, State),
  Index = ViewNumber rem length(Replicas),
  lists:nth(Index+1, Replicas).

set_client_requestnumber(ClientID, NewRequestNumber, State) ->
  ClientTable = get_from_list(clienttable, State),
  NewClientTable = lists:keyreplace(ClientID, 1, ClientTable, {ClientID
      , NewRequestNumber, -1}),
  set_in_list(clienttable, NewClientTable, State).

set_result(ClientID, RequestNumber, Result, State) ->
  LastRequestNumber = get_last_requestnumber(ClientID, State),
  ClientTable = get_from_list(clienttable, State),
  if
    RequestNumber == LastRequestNumber ->
      NewClientTable = lists:keyreplace(ClientID, 1, ClientTable, {
          ClientID, LastRequestNumber, Result}),
      set_in_list(clienttable, NewClientTable, State);
    true -> State
  end.

get_client_info(ClientID, State) ->
  ClientTable = get_from_list(clienttable, State),
  lists:keyfind(ClientID, 1, ClientTable).

is_duplicate(RequestNumber, ClientID, State) ->
  ClientTable = get_from_list(clienttable, State),
  {_,LastRequest,_} = lists:keyfind(ClientID, 1, ClientTable),
  if
    RequestNumber > LastRequest -> false;
    RequestNumber =< LastRequest -> true
  end.

get_last_result(ClientID, State) ->
  ClientTable = get_from_list(clienttable, State),
```

```
  {_,_,LastResult} = lists:keyfind(ClientID, 1, ClientTable),
  LastResult.

get_last_requestnumber(ClientID, State) ->
  ClientTable = get_from_list(clienttable, State),
  {_,LastRequestNumber,_} = lists:keyfind(ClientID, 1, ClientTable),
  LastRequestNumber.

update_clienttable(Log, State) ->
  Clients = get_from_list(clients, State),
  Counter = create_counter(length(Clients)),
  update_clienttable(Log, Counter, State).

update_clienttable([],_,State) -> State;

update_clienttable(Log, Counter, State) ->
  {_,Request} = lists:last(Log),
  {_,{_, ClientID, RequestNumber}} = Request,
  {_,IsUpdated} = lists:keyfind(ClientID, 1, Counter),
  Sum = sum_counter(Counter),
  Size = length(Counter),

  if
    Sum == Size-> State;

    Sum < Size , IsUpdated == 0 ->
      {_, OldRequestNumber, _} = get_client_info(ClientID, State),
      if
        RequestNumber > OldRequestNumber ->
          NewState = set_client_requestnumber(ClientID, RequestNumber,
              State);
        true -> NewState = State
      end,
      NewCounter = set_counter(ClientID, Counter),
      NewLog = lists:keydelete(Request, 2, Log),
      update_clienttable(NewLog, NewCounter, NewState);

    Sum < Size , IsUpdated == 1 ->
      NewLog = lists:keydelete(Request, 2, Log),
      update_clienttable(NewLog, Counter, State)
  end.

create_counter(Size) ->
  create_counter(Size, 1).

create_counter(Size, Counter) when Size >= Counter->
  [{Counter, 0}|create_counter(Size, Counter+1)];
create_counter(Size, Counter) when Size < Counter->
  [].

set_counter(Index, Counter) ->
  lists:keyreplace(Index, 1, Counter, {Index,1}).

sum_counter([]) -> 0;
sum_counter([Entry|List]) ->
  {_,Value} = Entry,
```

```
  Value + sum_counter(List).

ready_commit(OpNumber, State) ->
  WaitCommit = get_from_list(waitcommit, State),
  NewWaitCommit = lists:append({OpNumber}, WaitCommit),
  set_in_list(waitcommit, NewWaitCommit, State).

is_ready(OpNumber, State) ->
  WaitCommit = get_from_list(waitcommit, State),
  Tuple = lists:keyfind(OpNumber, 1, WaitCommit),
  if
    Tuple == false -> false;
    Tuple /= false -> true
  end.

commit(Central, OpNumber, State) ->
  CommitNumber = get_from_list(commitnumber, State),
  if
    CommitNumber+1 < OpNumber ->
      Ready = is_ready(CommitNumber+1, State),
      if
        Ready == true ->
          Log = get_from_list(log, State),
          Request = get_from_list(CommitNumber+1, Log),
          Result = serverservice:execute(Request, CommitNumber+1),
          vr_p:primary(replyPhase, State, OpNumber, Result),
          State2 = inc_value(commited, State),
          State3 = remove_commited(CommitNumber+1, State2),
          central:set_state(Central, State3),
          commit(Central, OpNumber, State3);
        Ready == false ->
          ready_commit(OpNumber, State),
          false
      end;
    CommitNumber+1 == OpNumber ->
      Log = get_from_list(log, State),
      Request = get_from_list(OpNumber, Log),
      {_,{_,ClientID, RequestNumber}} = Request,
      Result = serverservice:execute(Request, OpNumber),
      State2 = inc_value(commitnumber, State),
      State3 = remove_commited(OpNumber, State2),
      State4 = set_result(ClientID, RequestNumber, Result, State3),
      central:set_state(Central, State4),
      Result
  end.

remove_commited(OpNumber, State) ->
  WaitCommit = get_from_list(waitcommit, State),
  NewWaitCommit = lists:keydelete(OpNumber, 1, WaitCommit),
  set_in_list(waitcommit, NewWaitCommit, State).

commit_old_operations(Central, CommitNumber, State) ->
  MyCommitNumber = state:get_from_list(commitnumber, State),
  commit_old_operations(Central, CommitNumber, MyCommitNumber, State).

commit_old_operations(Central, CommitNumber, MyCommitNumber, State) ->
```

```
  if
    MyCommitNumber < CommitNumber ->
      Request = state:get_request(MyCommitNumber+1, State),
      {_,{_,ClientID,RequestNumber}} = Request,
      Result = server_service:execute(Request, MyCommitNumber+1),
      {_, LastRequestNumber, _} = state:get_client_info(ClientID, State
         ),
      if
        RequestNumber == LastRequestNumber ->
          State2 = state:set_result(ClientID, RequestNumber, Result,
             State);
        RequestNumber < LastRequestNumber ->
          State1 = state:set_client_requestnumber(ClientID,
             RequestNumber, State),
          State2 = state:set_result(ClientID, RequestNumber, Result,
             State1);
        true -> State2 = State
      end,
      NewState = state:inc_value(commitnumber, State2),
      commit_old_operations(Central, CommitNumber, MyCommitNumber+1,
         NewState);
    true ->
      State
  end.

add_to_log(Entry, State) ->
  Log = get_from_list(log, State),
  CommitNumber = get_from_list(commitnumber, State),
  Log2 = lists:append(Log, [Entry]),

  Rest = CommitNumber rem 15,
  case Rest of
    0 when CommitNumber > 0 ->
        NewLog = write_log_to_file(Log2, State),
        set_in_list(log, NewLog, State);
    _ELSE ->
        set_in_list(log, Log2, State)
  end.

write_log_to_file(Log, State) ->
  CommitNumber = get_from_list(commitnumber, State),
  ReplicaID = get_replica_ID(State),
  Name = string:concat("log", integer_to_list(ReplicaID)),
  {ok, File} = file:open(Name, [append]),
  NewLog = write_entry(Log, File, CommitNumber),
  file:close(File),
  NewLog.

write_entry([], _, _) -> [];
write_entry(Log, File, CommitNumber) ->
  [Entry|List] = Log,
  {OpNumber,_} = Entry,
  if
    OpNumber =< CommitNumber ->
      io:format(File, "~p ~n",[Entry]),
      write_entry(List, File, CommitNumber);
```

```
    OpNumber > CommitNumber -> Log
  end.

get_request(OpNumber, State) ->
  Log = get_from_list(log, State),
  get_from_list(OpNumber, Log).

is_primary(State) ->
  {_,Pid} = get_primary(State),
  if
    node() == Pid -> true;
    node() /= Pid -> false
  end.

get_replica_ID(State) ->
  Replicas = get_from_list(replicas, State),
  pos(node(), Replicas).

pos(_, []) ->
    not_found;
pos(N1,[{_,N1}|_]) ->
    1;
pos(N1,[_|Ns]) ->
    1+pos(N1,Ns).
```

Figura C.13: Módulo state.

```
-module(serverservice).
-export([execute/2]).

execute(Request, OpNumber) ->
  {request,{_,ClientID,_}} = Request,
  io:format("Executou a operacao numero ~p para  cliente ~p\n", [
     OpNumber, ClientID]),
  1.
```

Figura C.14: Módulo serverservice.

Bibliografia

[1] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. In *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[2] P. Bokor, J. Kinder, M. Serafini, and N. Suri. Efficient model checking of fault-tolerant distributed protocols. In *DSN'11*, pages 73–84. IEEE, 2011.

[3] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs, LNCS, vol. 5674*, page 23–42. Springer, 2009.

[4] M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: an overview. In *WS-FM'09*, pages 1–28. Springer, 2009.

[5] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in singularity OS. In *EuroSys'06*, pages 177–190. ACM, 2006.

[6] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[7] K. Honda, A. Mukhamedov, G. Brown, T. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT'11*, pages 55–75. Springer, 2011.

[8] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

[9] R. Hu, R. Neykova, N. Yoshida, R. Demangeon, and Honda. Practical interruptible conversations. In *RV'13*, pages 130–148. Springer, 2013.

[10] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[11] E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. Vasconcelos, and N. Yoshida. Specification and verification of protocols for MPI programs.

[12] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. In *IEEE Transactions on Software Engineering archive Volume 9*, pages 219–228. IEEE Press Piscataway, 1983.

[13] R. Stutsman and J. Ousterhout. Towards common patterns for distributed, concurrent, fault-tolerant code. In *HotOS'13*, pages 9–9. USENIX, 2013.